The Motor Industry Software Reliability Association

MISRA c/o Electrical Group, MIRA, Watling Street, Nuneaton, Warwickshire, CV10 0TU, UK.
Telephone: (024) 7635 5290. Fax: (024) 7635 5070. E-mail: misra@mira.co.uk Internet: http://www.misra.org.uk

Report 6

Verification and Validation

February 1995

PDF version 1.0, January 2001

Acknowledgements

The following were contributors to this report:

Main contributors:

Keith Hobley, The University of Leeds
Peter Jesty, The University of Leeds

Supported by:

Ian Kendall, Jaguar Cars
Keith Longmore, Lotus Engineering
David Ward, MIRA

Summary

A modern vehicle will be made up of many sub-systems some of which will use software. This report presents the verification and validation activities that should be performed upon these component sub-systems, especially the software sub-systems, and upon the vehicle as a whole. These verification and validation activities are presented in the order that they would normally be applied when adhering to a typical life-cycle. Specific methodologies and techniques are expanded upon in individual appendices.

Quality cannot be designed-in as an after thought and so early consideration of the issues highlighted in this report is essential.

The depth and rigour of the verification and validation activities will depend upon the integrity level of the system (see MISRA *Integrity* Report [1]). It is therefore prudent to identify the required integrity level as soon as the system is conceived in order to avoid unnecessary costs.

Final acceptance is based upon information concerning the system. The integrity level of the system will affect the nature and the volume of the required documentation, i.e. from formal reports, at the highest level, down to notes and comments, at the lowest level.
This document represents the current opinion within the community and as new techniques become accepted the recommendations made within this document should be augmented.

Recommendations

These recommendations are organised as they appear in the MISRA Guidelines.

Software life-cycle

Project planning

Planning definition

- Because of the problems associated with complexity, it is important that systems containing software are only used where their functionality and flexibility are required.
- The design team should produce a set of plans that, comprehensively and cost-effectively, cover the :
  - specification,
  - design,
  - programming,
  - integration,
  - verification and validation (including testing),
  - in-service support of the software.
- An appropriate life-cycle plan should be defined before the start of a project for the software, the system and the vehicle. It should ensure that verification and validation is performed throughout, with a rigour appropriate to the integrity level.
- The overall vehicle life-cycle should have clearly identified phases, and each individual system should also have its own life-cycle which should be compatible with the overall vehicle plan.
- The verification and validation activities should be defined to establish a level of confidence in the software which corresponds to that required for the integrity level of the system.
- Safety verification is specifically concerned with the safety requirements and should proceed concurrently with the normal design and implementation process. Each step of the development is also verified with respect to safety before progressing on to the next development phase.
- In order to confirm verification objectives, the translation from one development phase into the next should be analysed and/or inspected to ensure that each phase of the software development process faithfully maintains the attributes of the previous one without introducing errors.
- The purpose of software verification is to detect errors which may have been introduced during the software development process, and is concerned with the technical evaluation of the outputs of this process with respect to the inputs, including standards and guidelines.
- A software verification plan should be established covering each phase of the

software life-cycle.

- The purpose of validation is to determine whether the software satisfies its intention and mainly consists of performing tests, but it should also involve analysis and mathematical techniques where appropriate. Full validation of the software cannot usually take place until it has been integrated with the rest of the system and often the vehicle.
- To be effective, it is recommended that verification and validation is carried out by a person or team exhibiting a degree of independence from the design and implementation appropriate to the integrity requirements of the software and system. The degree of independence should be achieved by one or more of:
  - different person,
  - different department or section,
  - different company or division.

Assessment

- It is recommended that a company nominates a suitably qualified and independent assessor and/or auditor to perform product assessment
- Preparing for assessment is a task which should begin at the start of a project; this also avoids the possibility of arriving at the point of assessment and finding that a vital piece of information has been omitted or overlooked. Early liaison with the assessor is recommended.

Re-use

- "Off the shelf" components to be used in a project should be assessed to the required integrity level and such re-use of existing software should be considered on a case by case basis.
- Software that has performed faultlessly for a number of years in the field may well offer a greater level of confidence in the final product than a wholly new program, provided it is not modified, and is used within the limitations of its design.
- Software that is intended for re-use should have a standard of documentation such that its functions and interactions are easily understood by people who may not have been the original authors.

Requirements specification

Verification and validation of software requirements

- Team oriented reviews and analyses should be carried out to confirm that the software requirements specification is unambiguous, accurate, complete and can be verified.
- The software requirements specification should include the software functional requirements, the software non-functional requirements and the software safety

requirements. The latter should be specifically identified.

- The software safety requirements should describe the potential system failure conditions together with the functions which the software should perform as a result. They should also highlight any safety integrity requirements and design constraints.
- Safety integrity requirements should be divided into measures to avoid, and measures to control systematic and random failures; eg. requirements such as logical and physical partitioning, diversity, redundancy, condition monitoring and self-test.
- The software requirements specification should be verifiable. As a result the verification process may impose additional requirements or constraints on the software development process.
- Design constraints should be analysed to see if they are essential, unnecessary, or necessary but in the wrong form.
- The safety-invariants (statements of the safety of the system that should apply in all circumstances) are part of the system requirements specification.
- Each software high-level requirement should be traceable to one or more system requirements, and vice versa.
- Reviews of the preliminary safety assessment should be carried out as part of the project requirements review.

Tools and techniques for requirements specification

- An animation and/or a prototype can help the procurer and the supplier to assess the adequacy and correctness of the specification, and its use should be considered in order to stabilise the requirements before the design begins.
- If rapid prototyping is used there should be no attempt to use or modify the prototype code for production.
- Formal mathematical methods can help to ensure that a specification is unambiguous and well structured, and can support functional consistency checks, as well as providing a basis for sound review.
- A good formal specification should comprise a natural language description to support the precise mathematics to help those not familiar with the notation, and should read clearly with the mathematics omitted. The formal specification, however, should be the definitive text.
- If used, the application of formal mathematical methods should be supported by commercially available tools.

Design

Design for verification and validation

- Design for verification and validation, bearing in mind the capabilities and limitations of the available software tools, test rigs and in-vehicle aids.
- For a design to be comprehensible and maintainable, it must be constructed in a modular and structured manner.

- The software may have to be organised into sufficiently small modules to achieve the necessary test coverage.
- The definition of test coverage should be considered carefully.
- It should be possible to trace a module throughout a hierarchically described design; and to be able to relate the design to its specification.
- Design reviews should be carried out to establish that the design process is producing the required output at each relevant stage.
- Reviews of the detailed safety analysis should be carried out as part of the project design review.

Programming

Codes of practice

- Documented company and project standards should be used to give guidance on:
  - source code layout,
  - documentation style,
  - language usage,
  - compiler usage,
  - design best practice,
  - naming conventions.
- Code commenting is an integral part of any well prepared program and is vital for code verification.
- Consideration should be given to using a restricted subset of a programming language to aid clarity, assist verification and facilitate static analysis where appropriate.
- Changes to the software should be approved, fully documented and kept under a change control system.
- Due to the intangibility of the final program code, the production of supporting documentation is essential.

Verification and validation of code

- The documentation for the software modules should support the planned verification activities.
- Before dynamic testing begins the code must be reviewed in accordance with the software verification plan to ensure that it does conform to the specification.
- Code reviews and/or walkthroughs should be used to identify any inconsistencies with the specifications.
- All production and verification tools should be justified in relation to the integrity level of the software they are being used to analyse.
- Compilers should conform to an international standard for the language definition (where one exists), and if possible be validated for the target computer.

Programming tools and techniques

- If possible use automatic tools to check that company and project programming standards are being adhered to.
- The use of a checklist is useful in examining a program for common errors.

Testing

General

- The limitations of testing software based systems should be properly appreciated. The number of possible paths through a complete program means that even in the strictest test regime only a proportion of the paths will be executed.
- Testing forms one part of the overall verification and validation activities. It should be remembered that quality can never be tested into a product.
- The purpose of testing is to discover errors, not to prove correctness.
- Plan and document a test strategy, showing an intelligently selected test data set complete with justifications for the tests.
- A software validation plan should, for each functional requirement state the:
  - input conditions,
  - output conditions,
  - acceptance criteria,
  - limits of operation,
  - likely failure modes.

Static analysis

- Static analysis is effective in demonstrating a program is well structured with respect to its control, data and information flow. It can also assist in assessing its functional consistency with its specification.

Dynamic test

- The dynamic characteristics of the unit must be tested for compliance with the timing and response requirements, as well as the functional requirements.
- Consider both in-range and out-of-range values when selecting test cases.
- Test data should be derived from the software specification of the appropriate level, for example, requirements, design or module. It should remain consistent and traceable to that specification throughout.

System test

- System tests should occur in a realistic environment operating under realistic workloads.
- Confirm the effect on the system when operated under abnormal workloads

(stress testing).

- The system testing process should be based on the requirements. Coverage analysis should be used to show that the test cases are adequate to demonstrate compliance with each requirement.
- Test cases should be selected with the objective of uncovering errors. Previous experience should be used where possible (error guessing).
- Specific testing should be performed to determine whether or not the safety requirements are fulfilled.
- System tests should be performed using fully integrated, production intent software and hardware, in the real environment. Certain tests may require simulated environments.

Tools and techniques for testing

- Error seeding, may be used to demonstrate the effectiveness of error detection and recovery routines. This can give an indication of the detection rate of the testing process adopted.
- Equivalence partitioning can be used to reduce the test space by reasoning logically that not all possible combinations of test data are necessary.
- It may be useful to compare the results of testing the full implementation against the results provided by animating a formal specification (if possible) with the same test data.
- Boundary value analysis is a useful technique for the efficient testing of software, by selecting test inputs just above, just below and on the threshold values for key parameters.

Product support

Software maintenance

- Post-production modification should be performed with the same level of expertise, automated tools, planning and management as the initial development of the system. It is recommended that the procedures used for the product development are applied to post-production modifications.
- The replacement of a safety-related system by a new version should require the same levels of authorization and assessment, and follow the same procedures, as applied to the original system.

Software quality planning

Documentation requirements

- A Software Quality Plan should be established describing the overall approach to software development including a detailed life-cycle definition, the development environment, assurance controls, tools, standards, guidelines, and

the configuration management procedures used.

- As certain aspects of verification and validation will depend almost entirely on documentation it is vital that it is both concise and complete.
- The format and nature of the documentation produced will depend upon the size of the project, the integrity level and company policy. The information, however, should always be clearly identifiable.
- The handling of documentation should be properly regulated, both to control access to it, and to control changes.
- The period of time for which documents are to be retained should be laid down for, procedures defined for their control during that time, and plans made for their final disposal.

Emerging technologies

General

- New techniques, such as neural networks, object orientation and fuzzy logic, will have to judged individually against the confidence that can be placed on current techniques. The implications of using them in safety-related applications should therefore be considered carefully.

Neural networks

- Neural networks consist of a number of simple processors or neurons, linked together by connections or synapses. The neurons combine their inputs according to a set of weightings and subsequently produce an output which is passed to other neurons.
- The weights in the network are "trained" by applying sets of inputs and expected outputs to the network which iteratively adjusts the weighting factor of each connection.
- Once "training" is completed, the network should be switched to a "non-learning" mode. Leaving the network in its "learning" mode, for example to compensate for mechanical wear in a system, could permit unpredictable behaviour.
- Neural networks are a special case of adaptive control in which the network, by a process of feedforward prediction, "learns" about the device which it is to control, using a "learning" algorithm, and real recorded data.
- Neural networks are effective at handling non-linearities, and especially systems which have poor visibility for some, possibly critical, parameters.
- Neural networks may be very difficult indeed to validate.

Object orientation

- Object orientated techniques can be applied to any or all of the analysis, design and programming life-cycle phases. It views the software in a system as a set

of interacting objects with their own private states, rather than as a collection of functions. It is based on the idea of information hiding or abstraction.

- Once an object has been defined its interfaces should never be altered. This leads to easier maintenance and re-use.
- By identifying the main object classes in an application, object oriented approaches lend themselves to the creation of an object library in which commonly used object classes are maintained. This leads to efficient development process and to opportunities for re-use.
- It is worth noting that while there has been a great deal of interest expressed in object oriented techniques, they are much less mature than other equivalent techniques, and it still remains to be seen how effective they are in practice.

Contents

Page

Abbreviations

ALARP	As Low As Reasonably Possible
ASIC	Application Specific Integrated Circuit
BSI	British Standards Institution
CASE	Computer-Aided Software Engineering
DTI	Department of Trade and Industry
EMC	ElectroMagnetic Compatibility
EWICS	European Workshop on Industrial Computer Systems
DEFSTAN	Defence Standard
DRIVE	Dedicated Road Infrastructure for Vehicle safety in Europe
IEC	International Electrotechnical Commission
ISO	International Standards Organisation
MISRA	Motor Industry Software Reliability Association
MoD	Ministry of Defence
NCC	National Computing Centre
NPL	The National Physical Laboratory
PES	Programmable Electronic (sub-)System(s)
QA	Quality Assurance
RSRE	Royal Signals and Radar Establishment
STARTS	Software Tools for Application to large Real-Time Systems
UK	United Kingdom
VLSI	Very Large Scale Integration

Responsibilities List

Director	see Section 2.7 and Appendix B.
QA Manager	see Sections 2.6 and 8.5 and Appendix B.
Project Manager	see Sections 2.1, 2.2, 2.3, 2.6, 2.11, 5.6 and 7.3 and Appendices A, B, C.1, C.2, H and I.7.
Project Engineer	see Sections 2.3, 2.5, 2.9, 3.1, 3.2, 4.1, 5.3, 6.3, 6.4, 8.2 and 8.6 and Appendices B, C.1, C.2, F and G.
Software Manager	see Sections 2.4, 2.5, 2.8, 2.9, 2.10, 2.11, 3.1, 3.2, 3.3, 3.4, 3.5, 4.1, 4.2, 5.1, 5.3, 5.5, 5.6, 5.7, 5.8, 5.9, 7.1, 7.4, 7.5 and 8.9 and Appendices A, B, D.1, D.2, D.3, E.1, E.2, I.7, N.1, N.2, N.3 and O.
Software Engineer	see Sections 2.8, 3.3, 3.4, 3.5, 4.2, 4.3, 4.4, 5.2, 5.4, 5.7, 5.8, 5.9, 6.1, 6.2, 6.3, 6.4, 7.2, 7.3, 7.4, 7.5, 8.1, 8.2, 8.3, 8.4, 8.6, 8.7, 8.8 and 8.9 and Appendices B, D.1, D.2, D.3, E.1, E.2, F, G, H, I.1, I.2, I.3, I.4, I.5, I.6, J, K.1, K.2, K.3, L.1, L.2, L.3, M.1, M.2, M.3, M.4, N.1, N.2, N.3 and O.

Design Material Index

The following is a list of the various types of information that may be produced during the production of safety-related software. The page number in **bold** is where that design material is introduced. Note that in this PDF version, some of the page numbers may be inaccurate.

1. Introduction

Before software was incorporated into vehicles the testing and validation activities were normally performed upon the entire vehicle after the integration of all the sub-systems. However the flexibility and additional functionality provided by the addition of software to a vehicle's design increases the complexity to the extent that a more controlled development process is required, especially for safety-critical applications.

Requirements capture has emerged as an important stage in a system's life-cycle. If the user's requirements have been misinterpreted by the designers then ultimately the wrong system will be produced and there is a high cost associated with rectifying this discrepancy late in the project. Animation and prototyping are two of the techniques that can be utilised to reduce the risk of misunderstanding.

Testing must be performed throughout the software life-cycle. Both static and dynamic testing can be performed, but it should be noted that it is not normally possible for this testing to provide 100% coverage of the software.

Before the design and development teams move between life-cycle stages, reviews and analyses should be employed to increase the confidence that errors have not been introduced during the previous stage. One difference between an analysis and a review is that an **analysis** will involve the production of new material during the assessment process whilst a **review** involves assessing existing material. Both analyses and reviews involve the production of a feedback report.

The use of formal methods can enable a reviewer to apply a mathematically based argument or proof to verify the equivalence of two adjacent phases in the life-cycle. The degree of rigour employed in this process will depend upon the integrity level. It should however be noted that formal methods are no panacea and the choice as to their level of usage should be left to engineering judgement.

One of the central issues when trying to distinguish quality software from poor software is the lack of metrics currently available. Without the means of comparing the quality of two different sections of code, and thus without the means of justifying the statement "acceptable software is software that is good", we are forced to reduce this to the tautology "acceptable software is software that is accepted".

The final acceptance of the system will in part involve the scrutiny of any documentation or other forms of design material produced along with the software. As the integrity level increases the nature and volume of the documentation will be expected to change; for safety-critical systems (integrity levels 3 and 4) the information should normally take the form of formal reports whilst for a safety-related system (integrity levels 1 and 2) some of the information may take the form of notes or comments.

Throughout this report we refer to the documentation that is typically produced at each stage

in the life-cycle. It is intended that different documentation may be produced according to differing needs and Table 2.1 provides a matrix relating documentation against purpose. We refer to three forms of documentation; *plans* which are top-level documents sometimes used across projects, *specifications* which are specific "plans" designed for a particular project detailing how the plans are to be achieved, and *results* which record the outcome of performing the actions outlined in the relevant specifications. We also provide an index for each occasion that a document is referenced. It should be noted that it is the contents that are important rather than the existence of separate documents.

Various "good practices" for programming are emerging and each is intended to raise the confidence that any source code produced will conform to its requirements. Whilst some of these practices could be described as traditional, such as programming in a modular and structured manner, there are several techniques which are still in their infancy, such as object-orientation. For these less mature techniques it still remains to be seen how effective they are in practice.

Due to the cost involved in producing high integrity software there is a desire to increase the life-span of such software. Making the software re-useable and maintainable are both ways to achieve this goal.

Each topic is written with specific members of a company in mind. Herein we identify the *Director*, the *QA Manager*, the *Project Manager*, the *Project Engineer*, the *Software Manager* and the *Software Engineer*. These names are intended to refer to individual’s roles within a company and the actual job titles may vary between companies. To help an individual access those parts of this report pertinent to their activities we provide an index to the relevant sections.

2. General issues

2.1 A case for software verification and validation

Responsibility: *Project Manager*

Verification is concerned with the question "Are we building the system right?" and is performed by rigorously checking that the implementation of the system (or sub-system) meets its expected design. Verification is intended to be performed between each stage of a system's life-cycle, checking that each new stage faithfully adheres to the attributes of the previous stage. Software may thus be verified against its specification both with a procedure by procedure mathematical comparison against the *Software Requirements Specification*, and by tracing data through the design.

Validation of the other hand addresses the question "Are we building the right system?" and is a much more subjective process. During this phase we must check the completeness and consistency of all the initial requirements in the implementation, and also that no undesired functionality can occur. Since the initial requirements will normally be with respect to the entire vehicle, full validation of the software cannot usually take place until it has been integrated within the rest of the vehicle. Vehicle validation will justify the system's implementation by demonstrating that the propagated effects of the system faithfully meet, and do not violate, the initial vehicle requirements.

When a decision is made to use a Programmable Electronic Sub-system (PES) as part of a system it is essential that the nature of software is fully understood. Traditional engineering products can be considered as being evolved in two phases, development and production. Initially an idea is developed into one or more prototypes (see Appendix A). Once it has been confirmed that these prototypes do represent what is required, the production phase is entered which basically produces multiple copies of the same system. It is assumed that, once the prototype is approved, the design is "correct" and so, providing quality control is maintained during production, any failures will be due to wear or random faults in a component. This is the basis of Type Approval.

Unfortunately the development of software does not follow these stages in the same way. The production phase (producing multiple copies) is trivial and random faults due to wear are not applicable to software. Thus any faults that are in software appear during the interpretation of the requirements or during the development, and if they are not discovered they will remain as systematic faults in the production model.

The development of fault free software is notoriously difficult for two principal reasons:

- it is a digital system, and any state can follow any other state — the feature that provides flexibility,
- its potential for complexity — the feature that provides increased functionality.

These, of course, are the reasons for which it is used in the first place!

The complexity of software also means that it cannot, in general, be fully black box tested. Thus from the above argument the development process must be performed to a high standard if systematic faults are to be avoided.

The process of planning for the verification and validation must begin early on in the life-cycle, since the vehicle and its component systems must be designed in a manner that will enable them to be verified or validated.

As with all complex engineering systems particular attention must be given to the interfaces between sub-systems, especially if they are to be developed by different teams with different expertise and background. Systematic faults can occur due to omissions or misinterpretation of the interface requirements.

The additional complexity introduced by the use of software, and its associated hardware, mean that it should only be used when its advantages outweigh its disadvantages, especially in a safety-critical situation. The use of simple logic circuits, Application Specific Integrated Circuits (ASICs) or even custom designed Very Large Scale Integrated (VLSI) circuits can sometimes be just as effective, however it must be remembered that, since complexity is the issue, the development of complex ASICs or custom designed VLSI circuits can have comparable problems to the development of software.

The use of PESs within a vehicle can sometimes change the characteristics of that vehicle in a manner that is not fully understood by all potential drivers. It is therefore possible for safety hazards to occur due to an incorrect user model of the vehicle, even though there is no failure in a system.

2.2 Top level planning

Responsibility: *Project Manager*

2.2.1 Life-cycle definition

In order to ensure that the needs of safety-related software development meet the overall vehicle validation plan, it is helpful to identify the life-cycle of each sub-system. The life-cycle of a sub-system will depend on many factors such as:

- supplier (if bought-in),
- off-the-shelf, new or modified (for both hardware and software),
- previous experience (maturity),
- complexity of the system,
- integrity level of the system,
- risks associated with the system,
- quality assurance procedures,

- project timing constraints.

The overall vehicle development life-cycle should have clearly identified phases to handle the safety aspects, and each individual sub-system should also have its own life-cycle which must be embedded in, and compatible with, the overall plan. See Figure 2.1 for a typical example. A life-cycle defines the overall shape of a project, and should contain the development activities, their deliverables and reviewing mechanisms. The simplest form of life-cycle for software is the so-called "V-process" model, shown in Figure 2.2. This describes a software project in terms of a progression through a number of stages which recognises that the activities in the latter half of the project are all about testing the implementations of the specifications produced in the first half.

This simple model can be used where there is a good *System Requirements Specification*, a predictable output and few uncertainties. In most situations this is often not the case and the model should be refined to reflect the variations when applied to real system development projects. Some of the variations which have been identified are:

- *V-process with Prototyping Model* — here the simple traversing of the "V" is interrupted by prototyping. Prototypes aim to exhibit limited functionality to allow decisions to be made on such things as exact requirements, feasibility, or properties of the target environment. The V-Process model is useful where there are uncertainties or risks which require investigative work to resolve them. In planning the use of a V-Process model, the points at which prototypes are to be used should be identified, together with a definition of what the prototyping aims to achieve.
- *Evolutionary Development Model* — here the final system evolves through the use of a series of "V's", with each one producing a fully developed system with modifications. It is used where once a system is built, tested and implemented it is recognised that changes are required, these changes being included in the specification for the next system. The Evolutionary Development model is similar to the V-Process model except that each "prototype" is a completely developed system. Evolutionary Development models are often useful in projects where in-field experience and/or customer feedback are important factors; it does, however, often limit the potential to that of the first system.
- *Incremental Delivery Model* — here the final system is delivered through a series of "V's", the first one only giving core functionality, and each of the following passes adding more to this core until the full functionality is achieved. It is often used where project timing is a constraint and different functions can be identified as being required at different times on the critical path.

Few real projects will fall completely into any of the above life-cycle models, but will be a combination of these, with perhaps some special considerations unique to that project. However it should still be possible to define a life-cycle in detail, with all of the activities, their inputs, outputs, deliverables and reviews at strategically placed points during the

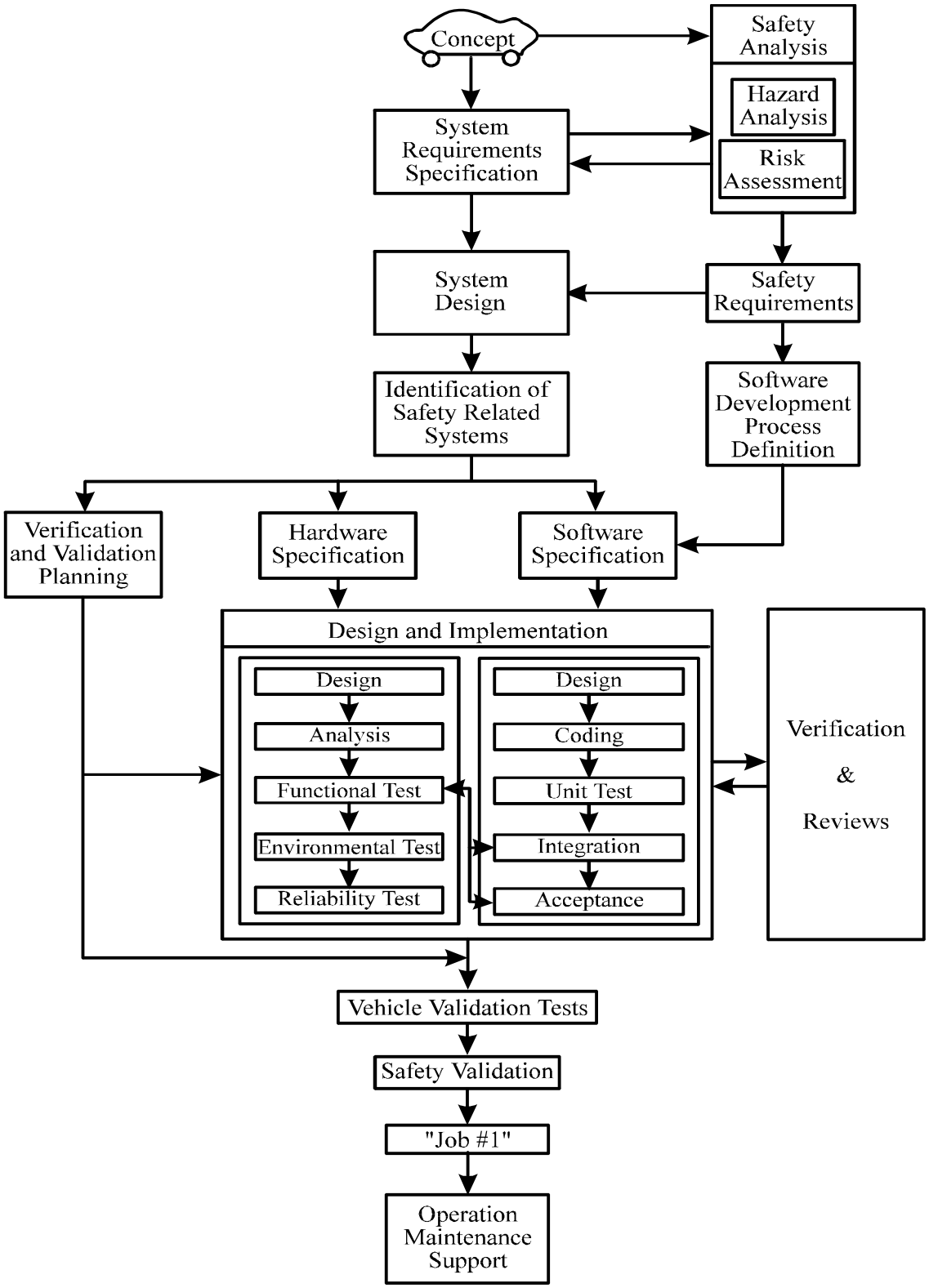


Figure 2.1 - System Life-cycle incorporating Software and Safety Life-cycles

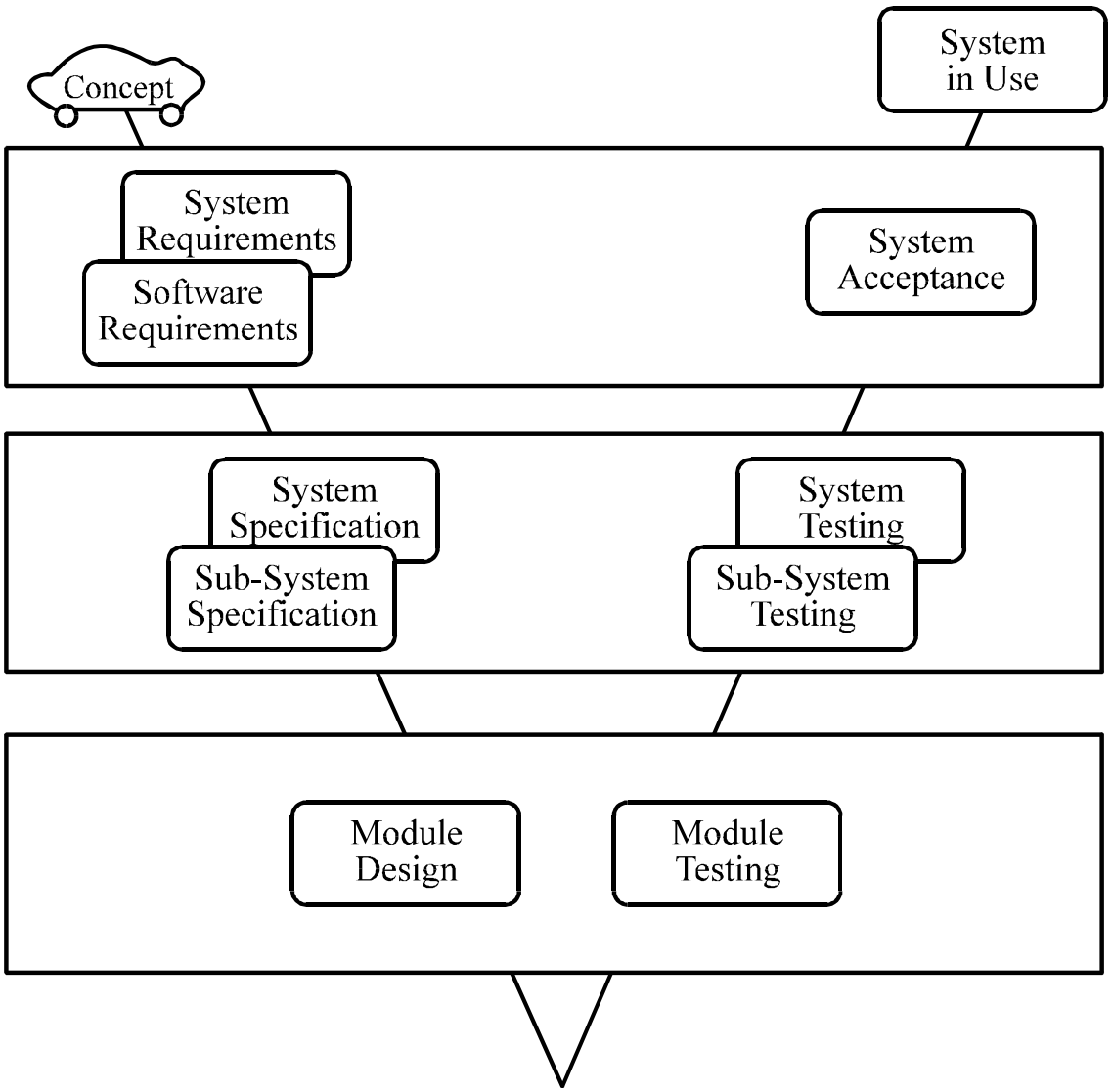


Figure 2.2 - Simple Software "V-process" Model

development. Life-cycles can be applied at all levels of the planning phase and are particularly important at the top level to ensure that the needs of the software life-cycle are reflected in the system and overall project life-cycles.

2.2.2 System aspects relating to software development

Although the MISRA Guidelines focus on software, it is impossible to ignore the impact of software on the system. After all it is the system that is the ultimate product, from which the driver of vehicle will derive benefit.

When considering any new vehicle feature which contains software, it is necessary to assess the whole system. It is not acceptable to say, "That is easy, we'll put it in the software", or even worse, "That's no problem we'll just change a bit of software".

We have already discussed some possible life-cycles for the system and the software, but we also need to recognise which factors are important as a result of the software when drawing up a top level system, or even complete vehicle, plan. Here we highlight those factors which must be accounted for by the project manager to ensure that those responsible for the software have the necessary support to do their jobs properly. A failure to recognise the importance of such advance planning will only lead to greater cost in the later stages of a project.

Some of the aspects of system design which may need thinking about at this stage are:

- accurate descriptions are required of the vehicle features which demand the use of a PES. This means that marketing and/or product definition departments must clearly state **what** task they want a feature to perform, and not simply state that a feature is required. This top level description is the starting point for the *System Requirements Specification* and the *Software Requirements Specification*, and should form an important part of the final validation tests,
- when deriving the *System Requirements Specification* from this description, often in the form of an initial system specification written by the project engineer, they must identify explicitly those requirements which are allocated to the software, especially those involving functionality, performance and safety. It may also be useful to refer to the hardware definition, to aid the software engineers to form a picture of the likely operating conditions of the system,
- where the integrity of the system depends on the software, this should be identified specifically. The potential system failure conditions should be described, together with the functions which the software should perform. This specification should also highlight any safety strategies and design constraints (see Appendix C) which are required, particularly aspects such as logical and physical partitioning, diversity, redundancy and self-monitoring.

2.3 Software validation planning

Responsibility: *Project Manager and Project Engineer*

Vehicle validation is the process of determining the level of conformity of the final operational vehicle to the initial requirements. Validation mainly consists of performing tests, but for software should also involve systematic analyses and mathematical techniques where appropriate.

Validation is concerned with demonstrating the consistency and completeness of a vehicle with respect to the initial requirements. As the initial requirements will normally be with respect to the entire vehicle, full validation of the software cannot usually take place until it has been integrated with the rest of the system and often the vehicle. (Despite this a number of validation tasks can usually be performed on the software before integration.)

The software should be exercised statically or dynamically using data present during normal operation, anticipated occurrences when the system reaches, or is near, its limits, and fault conditions requiring system action.

The *Software Validation Plan* should state for each required function, or other requirements:

- the required input data with their sequences and their values
- the anticipated output with its sequences and values
- the acceptance criteria
- the expected limits of operation
- the likely failure modes.

It should also specifically address each safety function, such that each is confirmed by representative tests. It is recommended that these tests:

- cover all signal ranges, and the ranges of computed or calculated parameters in a fully representative manner
- cover the logic combinations comprehensively
- cover timing requirements in a representative manner
- ensure that accuracy and response times are confirmed
- provide all typical displays to the users
- provide for all typical sequences of output signals
- ensure that correct action is taken for any equipment failure or failure combination stated in the requirements
- provide a rigorous correctness argument on the code (static tests).

In certain circumstances the *Software Validation Plan* may include a large number of test cases with the results being evaluated probabilistically. Exceptionally, previous operating experience may be included in the *Software Validation Plan* provided such experience is evaluated carefully and supported by appropriate probability theory Any statistics used should be validated by a qualified person, especially for the higher integrity levels.

It is recommended that validation is carried out by a person or team exhibiting a degree of independence from the design and implementation appropriate to the integrity requirements of the system (see Section 2.7).

Other related design material:

- the *Test Plan*.

2.4 Software verification planning

Responsibility: *Software Manager*

The procedure for software verification is concerned with the technical assessment of the outputs of the software development process with respect to the inputs, including standards and guidelines. It is a general measure to examine the translation from one development stage into the next by inspection, and to ensure that this process faithfully maintains the attributes of the previous stage. In order to assure the quality of the final software product the verification process should be performed in accordance with a *Software Verification Specification*. The degree of effort required to meet the verification objectives will vary according to the integrity level required of the development process.

It should be noted that verification is not simply concerned with testing, since testing, in general, cannot show the absence of errors.

Safety verification is specifically concerned with the safety requirements and should proceed concurrently with the normal design and implementation process, so that each step of the development is also verified with respect to safety before progressing on to the next development phase.

To be effective, it is recommended that verification is carried out by a person or team exhibiting a degree of independence from the design and implementation appropriate to the integrity requirements of the software, and must take place throughout the software life-cycle (see Section 2.7 and MISRA *Integrity* Report [1]).

2.4.1 Verification objectives

The objective of software verification is to detect errors which may have been introduced during the software development processes (elimination of these errors is left to the development processes). Specifically, verification serves to determine that:

a) The *System Requirements Specification* allocated to software have been properly developed into the high-level *Software Requirements Specification*.
b) The high-level *Software Requirements Specification* have been properly developed into the software architecture and low-level requirements. In cases where one or more levels of requirements are developed between high-level, and low-level requirements, proper development of the successive levels of requirements should also be

determined. In cases where code is generated directly from high-level requirements, this objective does not apply.

c) The software architecture and low-level requirements have been properly developed into source code.

d) The code in executable form complies with the *Software Requirements Specification*.

e) The means used to meet the above objectives are technically correct and complete for the required software integrity level.

f) Errors found during the verification process are be fed back to the software development processes.

2.4.2 Verification activities

To meet the above objectives, the *Software Requirements Specification* should be verifiable. Requirements verifiability and completeness may be determined by developing test cases early in the requirements process. As a result, the verification process may impose additional requirements or constraints on the software development process.

Verification objectives are met through a combination of reviews, analyses, the development of test cases and procedures, and the subsequent execution of those test procedures.

Reviews and analyses are used to assess the accuracy, completeness and verifiability of the requirements, architecture and source code. The development of test cases may provide further assessment of the internal consistency and completeness of the requirements.

As part of the verification process suitable test cases should be run on the executable object code. It follows that the code verified should be the same as that embedded in the final system.

The means used to meet verification objectives should itself be verified by reviews, by test coverage analysis and/or by tool qualification.

In some cases, it may not be possible to verify specific requirements by exercising the software in a realistic test environment. Such specific cases should be identified in the *Software Verification Specification* and an alternative means should be defined and justified.

Traceability data provides evidence of both the implementation of the requirements and the associated verification of those requirements. Traceability can be demonstrated by defining the data during the software development process, or by conducting an analysis as part of the verification process, and then documenting the traceability data. The products of the traceability analyses and test coverage analyses should demonstrate that each requirement is traceable to the code which implements it, and to the relevant test cases.

2.4.3 Impact of software integrity level on verification objectives

The degree of effort required to meet the verification objectives varies according to the software integrity level as follows (see also [1]):

a) High-level requirements, and traceability to those high-level requirements, should be verified for all software levels.
b) For the higher software integrity levels greater emphasis is placed on:
   i) Verification of low-level requirements.
   ii) Verification of the software architecture.
   iii) Degree of test coverage.
   iv) Control of the verification procedure.
   v) Independence of the verification activities.
   vi) Verification activities which overlap each other in terms of error detection.
   vii) Robustness testing.
   viii) Verification activities which have an indirect effect on error prevention or detection (e.g. conformance to standards or guidelines).

2.4.4 Inputs to, and outputs from the verification planning process

The inputs to the verification planning process are:

a) The *Software Requirements Specification*.
b) The *Software Safety Requirements*.
c) The software architecture.
d) The software integrity scheme.
e) Applicable standards and guidelines.
f) The quality assurance approach.
g) The configuration management approach.
h) The development approach.
i) The verification approach, including the use of software tools.

The outputs of the verification planning process are:

a) The *Software Verification Specification*.
b) Feedback and changes to the development life-cycle.

2.4.5 Verification planning

Verification is concerned with the confirmation, or otherwise, of the self consistency of the various stages in the design and implementation phases. Each stage of the software development life-cycle should have a *Software Verification Specification* established for it. Care should be taken to ensure that the product, as it is developed, complies with each design requirement. Various methods can be used, either in isolation or in combination with other methods:

- analysis
- simulation
- emulation
- modelling
- calculation

- demonstration
- inspection
- test.

The *Software Verification Specification* should document all the criteria, techniques and tools to be utilised in the verification process at each stage. It should describe the activities necessary to evaluate each item of software, and to demonstrate whether the system's functional, reliability, performance and safety requirements are met in accordance with the *Software Requirements Specification*. It should also state the test items and their versions required (e.g. a prototype) or the activities to be performed at a pertinent stage of the project (e.g. when a mathematical model is to be used during the development). The *Software Verification Specification* should not give detailed test procedures, but only state when a test will be done and what requirements will be verified by that test. If it is complete, it can be used as a checklist to demonstrate that compliance with all the requirements is or will be proved.

The level of detail should be such that the independent person or team can execute the *Software Verification Specification* and reach an objective judgement as to whether or not the software is adequate.

The *Software Verification Specification* should be considered as a set of guidelines and templates for generating verification procedures, the contents may vary with the software integrity level.

The specification should include, but is not limited to:

a) Compliance with objectives: A statement of how the verification objectives are to be satisfied for the project.
b) Organisation: Organisational responsibilities within the verification process and interfaces with the other software life-cycle processes. The approach for establishing verification independence, when required, should be described (see Section 2.7).
c) Verification methods: The verification methods to be used for each step of the verification process should be specified. Review methods should be specified. Analysis methods should be specified. Guidelines for both requirements-based and any structure-based tests should be specified which establish the test case selection process, the types of test procedures to be used, and the test data to be produced.
d) Verification environment: The equipment to be used for testing, as well as any testing or analysis tools to be used in verification should be specified and be traceable.
e) Acceptance criteria: The acceptance criteria for moving from one stage in the development life-cycle to the following stage.
f) Partitioning considerations: If partitioning is used in the software, then the method(s) used to verify the integrity of the partitions should be specified, since the basis for the entire design of a system would be invalidated if the partitions could not be verified to the appropriate level. ie. The overall software integrity depends on that of its constituent modules and the data and control flows between them.
g) Compiler assumptions: Any assumptions about the correctness of the compiler, linker

or loader should be included.

h) Re-verification guidelines: If the software is to be changed, the methods to be used to identify the affected areas of the change, and the methods to identify all modified object code should be specified. The re-verification criteria should allow an orderly recovery from errors or classes or errors which may be detected, including those discovered during the software integration and system integration testing.

i) Previously developed software: For previously developed software if the initial compliance baseline is not based on these guidelines, the methods for achieving verification equivalent to the MISRA Guidelines should be specified.

j) Multi-version software: Where multi-version software is used, related verification activities should be specified.

2.4.5.1 Software verification cases and procedures

Software test cases should identify the sets of inputs, conditions and expected results for each test to achieve the objectives in the Software Verification Specification. The verification procedures should detail step-by-step how each test is to be set up, executed, and the results evaluated. This data should include, but is not limited to:

a) The exact purpose of each verification activity.
b) How each verification activity is to be implemented.
c) The environment and tools to be used.
d) The specific pass/fail criteria.

2.4.5.2 Software verification results

The Software Verification Results are the outputs produced by the verification activities. The verification results of the reviews, analyses and tests should identify each procedure that failed during verification as well as the final pass/fail result. The results should be easily identifiable with the procedure conducted to generate them and with the software version which generated them. The results of any required coverage and traceability information analyses should also be shown.

Other related design material:

- *Software Design Verification Results*

2.4.5.3 Qualification criteria for software tools

Whist the developer will be principally concerned with verifying and validating the application program(s), it must not be forgotten that the development process will itself use other programs as tools (eg. compilers, CASE (see Section 5.6)). These are themselves complex programs which may introduce errors. There are currently no formally developed and proven tools commercially available.

Conformance tests can be performed on compilers to demonstrate that they conform to the language standard, and evaluation centres also exist that provide further assessment relating

to their fitness for purpose. Whilst they cannot be relied upon to establish the correctness of the object code in all circumstances, validated compilers should be used whenever possible and especially for the higher integrity levels.

Whilst one can expect the situation to change in the long term, until then developers of safety-related software must be aware of the problem and choose software tools with care, recognising that they are also potential sources of faults.

2.5 Analysis and review procedures

Responsibility: *Project Engineer and Software Manager*

Reviews and analyses are important tasks in the verification process, as they contribute to the prevention and elimination of errors. One distinction between a review and an analysis is that an analysis provides repeatable evidence, and a review provides a qualitative assessment of correctness. A review may consist of an inspection of an output of a process guided by a checklist or similar aid, whereas an analysis may examine in detail the functionality, performance, traceability and safety implications of an item of software, together with its relationship to other items within the system, and may include the use of formal methods.

Reviews may be required:

a) At defined milestones, prior to progressing to the next phase.
b) To consider major design issues.

These reviews will in general aim to establish that proper and adequate development procedures have been adhered to, including software design standards appropriate for the project; that the results of the analyses are acceptable, or that identified actions arising from the analyses or tests results are, or have been, carried out correctly; and that the design fulfils the requirements of the *Software Requirements Specification*.

2.6 QA issues

Responsibility: *QA Manager and Project Manager*

The MISRA Guidelines recommend that all companies should work to a recognised quality system standard such as ISO 9000/BS 5750 or equivalent. This will prescribe the general requirements of a quality orientated organisation and the attributes necessary for the assurance of that quality. The usual situation is the existence of a dedicated function within the organisation. The Quality Assurance (QA) department will be expected to have a person or persons responsible for the QA, and importantly, have the authority and freedom to ensure that the requirements of the standard are met and to implement corrective action if they are not.

Software should also be part of the responsibility of the quality assurance function and software quality planning is a key factor of this. The software part of the *Quality Assurance Plan* should be produced by the QA department to run in parallel with the other quality plans which are required to meet the standard (the TickIT guide [2] can help in the application of BS 5750 to software quality). It should outline the disciplines which must be carried out during the development of software and is likely to include:

a) A description of the overall development approach, including an outline of the generic software life-cycle, the development environment and tools used.
b) A definition of the outputs of each phase and the controls which are to be used to assure these outputs, including reviews, testing, etc.
c) A list of appropriate standards to be used for requirements, design and programming.
d) A description of the configuration management procedures covering:
   - the conventions for the unique identification of software items and associated documents (including variants and versions),
   - procedures for change control ensuring that changes are tracked and fed back into the appropriate part of the life-cycle,
   - methods of controlling, storing, copying (including backing-up) and transit of software items and associated documents,
   - status reporting,
   - indexing of all identified items and documents.

It is also the responsibility of QA department personnel to monitor conformance against the *Quality Assurance Plan*, and to implement corrective action in the event of non-conformance.

The implication of this on most automotive companies will be the deployment of QA personnel with a suitable software background, or training of current staff, to take on the extra role of responsibility for software quality. This applies equally to suppliers and to vehicle manufacturers.

2.6.1 Quality management plan

This is a high level document which identifies the software quality control functions and activities, and assigns specific responsibilities and authorities to ensure that they are carried out. It should identify named individuals (or named positions) who would be accountable for decisions, and it should also satisfy an assessor that the software quality control manager has the authority to act independently and to resolve problems.

The *Quality Management Plan* should also include plans for periodic and systematic reviews of the software quality control system.

2.6.2 Quality assurance plan

Assurance of the correctness of a system has to be made by an assessment of the procedures used in the design and implementation (the development process) of the system, and how the development, production, distribution and maintenance processes are controlled. The degree

of independence and authority of the assessment process is the crux of the matter and is dependent upon the integrity level (see ST2 Integrity).

This is recorded in the *Quality Assurance Plan*.

Other related design material:

- *Configuration Plan*

2.7 Independence

Responsibility: *Director*

The issue of independence arises in relation to the verification and validation of safety-related systems due to the emotive nature of anything to do with safety. Not only must a system have been developed correctly, but there is an increasing demand that it should be **seen** to have been developed correctly. No reputable company will have a policy for creating unsafe products, nor will the professional code of conduct permit an engineer to knowingly embark on an unsafe practice. Nevertheless experience has shown that under certain circumstances management pressure can be such that the unthinkable can occur (e.g. a primary cause of the Challenger Space Shuttle disaster[1]). It is in order to alleviate this latter scenario that some form of independence is considered to be necessary in the verification and validation process.

The degree of independence required will depend upon the integrity level to be achieved (see [1]). IEC/SC65A/WG9 and IEC/SC65A/WG10 identify three degrees of independence as follows:

- independent organisation
- independent department
- independent person.

The independence is classified in terms of having separate and distinct financial, management and other resources, from the organisation, department or persons responsible for the main development of the system. However they also state that those companies that have internal organisations skilled in risk assessment and the application of safety-related systems, which are independent of and separate from those responsible for the main development, may be able to use their own resources, even at the highest integrity level. If such internal organisations do not exist then, if it is not possible to set one up, the requirement for independence may have to be met by using an external organisation.

[1] Although the safety hazard had been identified, the requirement to keep the flight on schedule and thus maintain credibility, meant that the warnings were not fully heeded.

2.8 Design material and documentation

Responsibility: *Software Manager and Software Engineer*

The design team should produce a set of design material that, comprehensively and cost-effectively, covers the specification, design, integration, verification and validation, and in-service support of the software. Each document should be well structured with a glossary of locally defined terms, and be identified in accordance with the *Configuration Plan* section of the *Quality Assurance Plan*. A higher level of assurance can be obtained by using a method such as the Standard Generalised Mark-up Language (ISO 8879) to structure the documents; use of a defined, restricted vocabulary may also be beneficial. Representative examples of the proposed design material should be agreed between the design team and the assessor (see [1]) at the start of the project.

The design material should enable the complete development environment to be re-established by another contractor for the purposes of maintaining the software. It should therefore identify the precise versions of support software and hardware, and give all necessary details such as job control commands used to start building the software.

The design material should include a copy of the relevant parts of the *Software Requirements Specification* and provide details of any amendments and concessions.

2.8.1 Design material required

There are two aspects to the requirements for design material:

- what design material is necessary?
- when it is needed?

Many generic standards and guidelines exist that define documentation requirements for all aspects of the software development life-cycle (see [3] and [4]) and some of the large automotive companies have developed and are using their own standards.

The ISO/TC22/SC3/WG11, Automotive Electronic Control Systems — Technical Documentation working party is producing a standard for documentation interchange between vehicle manufactures and component suppliers. This work will incorporate proposals from the German automobile industry consortium, MSR, and will conform to the requirements of the draft EEC Directive 70/156/EEC. The ISO standard is likely to be available in draft form by mid-1994.

Therefore, it is inappropriate to define a new or alternative documentation regime at this time and the adoption of the new ISO documentation standard for Automotive Electronic Control Systems is recommended. However, we do suggest a list of design material that is typically produced during the development of a product. Due to the invisibility of the final program code, the production of supporting design material is essential. This design material is required for different reasons:

- to help assure a disciplined approach to the software life-cycle,
- to provide detailed information about the product sufficient for any future maintenance and/or re-use,
- to provide evidence of the software quality for assessment.

Table 2.1 is a matrix relating possible design material against these three purposes. The titles indicate the type of information required, rather than the necessity for separate reports. The number of bound volumes produced will depend both on the size of the project and on company policy; the information, however, should always be clearly identifiable. See Appendix B for an example of the contents of these documents.

2.8.1.1 Integrity level

There is an accepted principle that the greater the risk, the greater is the need to supply information and to give evidence of robustness. Therefore, the integrity level of the system (see [1]) will dictate the amount and the form of the information provided. For the assessment of safety-critical systems (integrity levels 3 and 4) the information should normally take the form of formal reports. For a safety-related system (integrity levels 1 and 2) some of the information may take the form of notes or comments e.g. commenting within the code. For a non-safety system (integrity level 0) only that information necessary for the organisation's usual *Quality Assurance Plan*, e.g. the TickIT guide [2], will be required as formal reports.

Throughout this report we will use the names Plan, Specification and Results in a strict sense, namely:

- *Plans* — Top-level information which can sometimes be used across projects. Plans look forward to future activities.
- *Specifications* — A specific "plan" designed for a particular project. It will typically contain details of how the plans are to be achieved.
- *Results* — A record of the outcome of performing the actions outlined in the relevant specification. Results look backwards at past activities.

2.8.2 Software assessment information

The software design information should be assessed against the requirements of an approved quality management system, such as ISO 9000-3 (BS 5750 Part 13). It is not a formal requirement for the supplier to hold ISO 9000 (BS 5750) or equivalent accreditation, but that the information on the software is of equivalent or greater quality.

The MISRA Guidelines should be used as guidance for the assessment, though it is recognised that the software may have been produced under other acceptable standards (e.g. IEC/SC65A/WG9).

The integrity level (see [1]) should define the depth of software analysis required and should also identify the level of design approval required for the information on the software. A

Information required	Life-cycle	Maintenance	Assessment
Software Design Plan	✓		✓
Software Validation Plan	✓		✓
Software Design Review Plan	✓		✓
Software Acceptance Plan	✓		✓
Software Verification Specification	✓	✓	✓
Software Requirements Specification	✓	✓	✓
Software Requirements Verification Results	✓		✓
Software Detailed Design Specification	✓	✓	✓
Software Module Specification	✓	✓	✓
Software Design Verification Results	✓		✓
Software Module Design Verification Results	✓		✓
Software Design Review Results	✓		✓
Software code and supporting documentation	✓	✓	✓
Code Review Results	✓		✓
Software Acceptance Test Specification	✓	✓	✓
Software Module Test Specification	✓	✓	✓
Software Integration Test Specification	✓	✓	✓
Software Module Test Results	✓		✓
Software Module Error Incidence Results	✓		✓
Software Integration Test Results	✓		✓
Software Integration Error Incidence Results	✓		✓
Software Validation Results	✓		✓
Software Acceptance Test Results	✓		✓
Maintenance Procedures (see Figure 8.1)	✓	✓	✓
Quality Management Plan	✓	✓	✓
Quality Assurance Plan	✓	✓	✓
Owner Handbooks	✓	✓	✓
Service Manuals	✓	✓	✓

Table 2.1 - Relation between design material and purpose

formal assessor's report will be required where this is deemed necessary by the integrity level defined for the software.

The information on the software verification and validation should meet the requirements of the integrity level. A suitable "checklist" will greatly assist in the initial evaluation of the information on the software.

2.8.3 Owner handbooks

These refer to any documents that accompany and explain how to use the product.

2.8.4 Service manuals

These refer to any documents that explain how to maintain the product.

2.9 Relation to life-cycle stages

Responsibility: *Project Engineer and Software Manager*

Figures 2.3 and 2.4 provide a structure as to how the information required in Section 2.8.1 is related.

2.10 Control

Responsibility: *Software Manager*

The objective of control is to ensure that the handling of information is properly regulated. There are two aspects to control:

- control of access: to ensure that only properly authorized personnel have access to the information. This is particularly important where the information contains sensitive material.
- control of change: to ensure that all changes are properly managed.

The MISRA Guidelines assume that companies have in place Quality Management Systems capable of meeting the requirements of ISO 9000-3, such as the TickIT guide [2]. ISO 9000-3 contains a section on document control, which should be the starting point for a supplier's information control system.

2.10.1 Control of access

Control of access ensures that only properly authorized personnel have access to the information. The main motivations are:

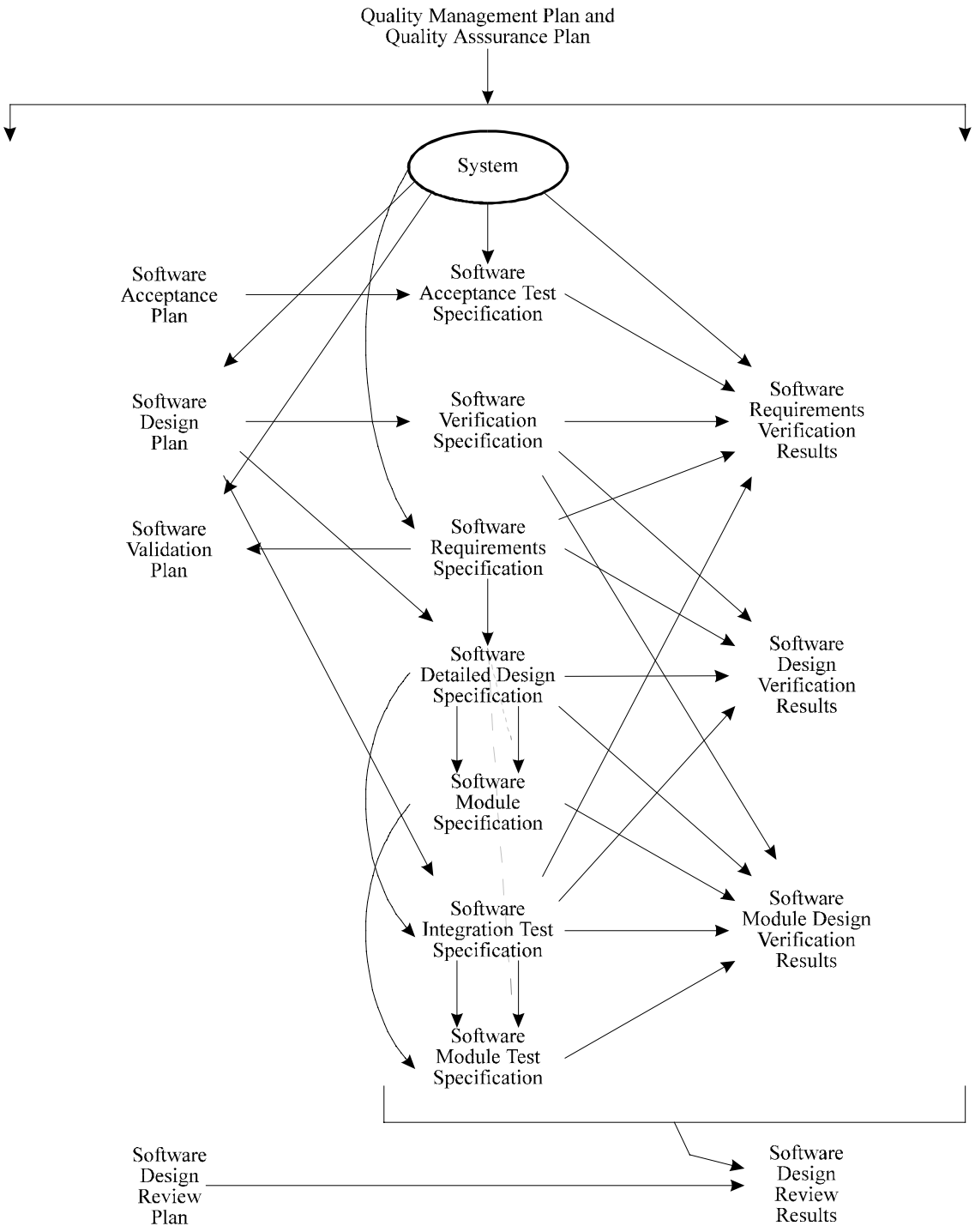


Figure 2.3 - Design Material Interrelationship 1

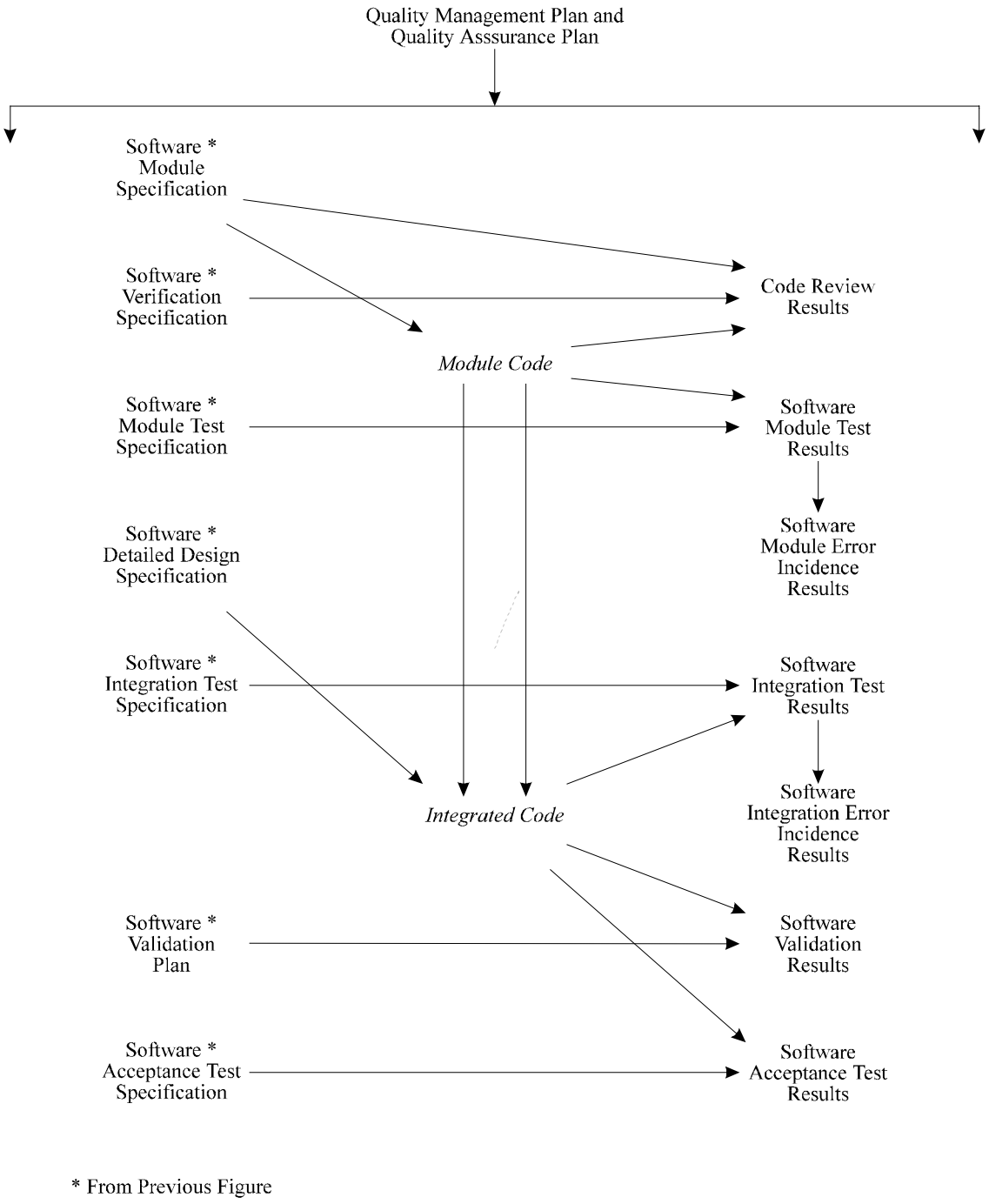


Figure 2.4 - Design Material Interrelationship 2

- confidentiality: to ensure that proprietary information is not disclosed, for example where a sub-contractor has been used to provide software and does not want the source code to be disclosed.
- security: information contained in documents may be used for nefarious purposes, such as "chipping" or illicit access to diagnostic and service mechanisms.

Access control can be achieved by the use of:

- secure filing systems (both paper based and electronic),
- lists of authorized users,
- procedures controlling access and modification of authorization lists,
- audit trails listing access attempts, both successful and unsuccessful,
- passwords and encryption for information stored in machine-readable format.

Certain documents may have to be vested in another party (see escrow in [5]). Source code is an example, where the producer of the software does not wish to hand proprietary information over to the customer, but the customer needs guaranteed availability of the source code in the future, should maintenance be required.

2.10.2 Control of change

2.10.2.1 Procedures

To avoid incorrect changes being made to documents, or changes being made to incorrect versions of documents, procedures should be established to control all documents. These procedures will need to cover:

- the scope of change control, including a list of documents that fall within its remit,
- approval and issuing of further procedures relating to document control,
- release, updating and withdrawal of documentation.

2.10.2.2 Audit trail

An audit trail of modifications should be established and maintained. The audit trail is an object based log which enables identification of all changes and the responsible party. Whilst the technique is primarily applicable to machine readable files, and indeed to software itself, it is also applicable to paper based documentation. An audit trail record needs to cover:

- identification of the document
- release level to which the modification is applicable
- date and time of change
- person responsible
- reason for change
- authorization (including justification if no authorization required)

- details of the change. This should take the form of a summary, indicating the location within the document of the changes.

2.10.2.3 Modification records

All documents should contain a modification record, giving:

- unique identifier, which should tally with the audit trail
- date
- responsible person(s)
- summary of change
- location of change.

The change itself should be clearly marked within the text, for example by using change bars.

2.10.2.4 Version numbers

Procedures should define the criteria for a new version of a document being issued. Typically, this will be when a certain number of changes have been made or when a fundamental revision occurs. The issue of a new version may require appropriate authorization and justification. The procedures should also define the retention policy for old versions of the document.

2.11 Leading to assessment

Responsibility: *Project Manager and Software Manager*

One of the objectives for maintaining a documented record of the product is to prepare for an assessment process. For this activity it is essential that the documentation is complete, correct and readily accessible to the assessor (but note Section 2.10.1). This is true for all forms of assessment; first party. second party and third party (see Section 7.2).

Preparing for assessment is a task which must begin at the start of a project, otherwise the possibility exists of arriving at the point of assessment and finding that a vital piece of information has been omitted or overlooked. Early liaison with the assessor is recommended to ensure the actual assessment is as smooth as possible and that all the assessor expects to find has been completed.

In preparation for assessment, the following should be generated:

- list of required information (see Section 2.8.1). If certain information is omitted from a particular project or phase this should be justified, along with the appropriate authority. Similarly, any **additional** information needed should be listed
- checklist of documents, covering:

- name of document
- responsibility
- required date
- received date.

In addition, the information should be reviewed prior to submitting for assessment. The checklist should contain a record of this being done.

The access and availability of the information should be reviewed to ensure it is readily accessible by the assessor, particularly in the case of machine-readable information. However, the security of the information must not be compromised in any way (see Section 2.10.1).

3. Software requirements specification

3.1 Design material

Responsibility: *Project Engineer and Software Manager*

In order to clarify the following discussion the relationship between the various types of information are shown in Figure 3.1, and are each briefly described below.

System Requirements Specification — a structured document which sets out the system services in detail. It is divided into the *System Functional Requirements*, the *System Safety Requirements* and the *System Non-Functional Requirements*.

System Functional Requirements (and *System Functional Safety Requirements*) — the functions that the system must perform to satisfy the operational requirements and the safety requirements of the system. They consist of high level requirements, which can be developed into low level requirements, together with any derived requirements (additional requirements resulting from the development processes which are not directly traceable to higher level requirements) that have been identified.

System Safety Requirements — these are divided into the *System Functional Safety Requirements* and the *System Safety Integrity Requirements*.

System Non-Functional Requirements — the non-functional characteristics required of the system.

System Functional Safety Requirements — the functions that must be carried out by the safety-related control system should a hazardous failure occur in the technical process or in the safety-related system itself.

System Safety Integrity Requirements — measures to ensure that all safety functions are performed in order to meet the integrity level. They are divided into measures to avoid and to control systematic and random faults.

Software Requirements Specification — the software aspects of the *System Requirements Specification.* It may also contain constraints on the software. It is divided into the *Software Functional Requirements*, the *Software Safety Requirements* and the *Software Non-Functional Requirements*.

Software Functional Requirements (and *Software Functional Safety Requirements*) — the functions that the software must perform to satisfy the operational requirements and the safety requirements of the system. They are deduced from the *System Functional Requirements* and the *System Functional Safety Requirements*, respectively.

Software Safety Requirements — these are deduced from the *System Safety Requirements,* and

are divided into the *Software Functional Safety Requirements* and the *Software Safety Integrity Requirements*.

Software Non-Functional Requirements — the non-functional characteristics required of the software.

Software Functional Safety Requirements — the functions, allocated to software, that must be carried out by the safety-related control system should a hazardous failure occur in the technical process or in the safety-related system itself.

Software Safety Integrity Requirements — the measures to achieve the necessary dependability to meet the integrity level. These form part of the *System Safety Integrity Requirements*.

3.2 System description

Responsibility: *Project Engineer and Software Manager*

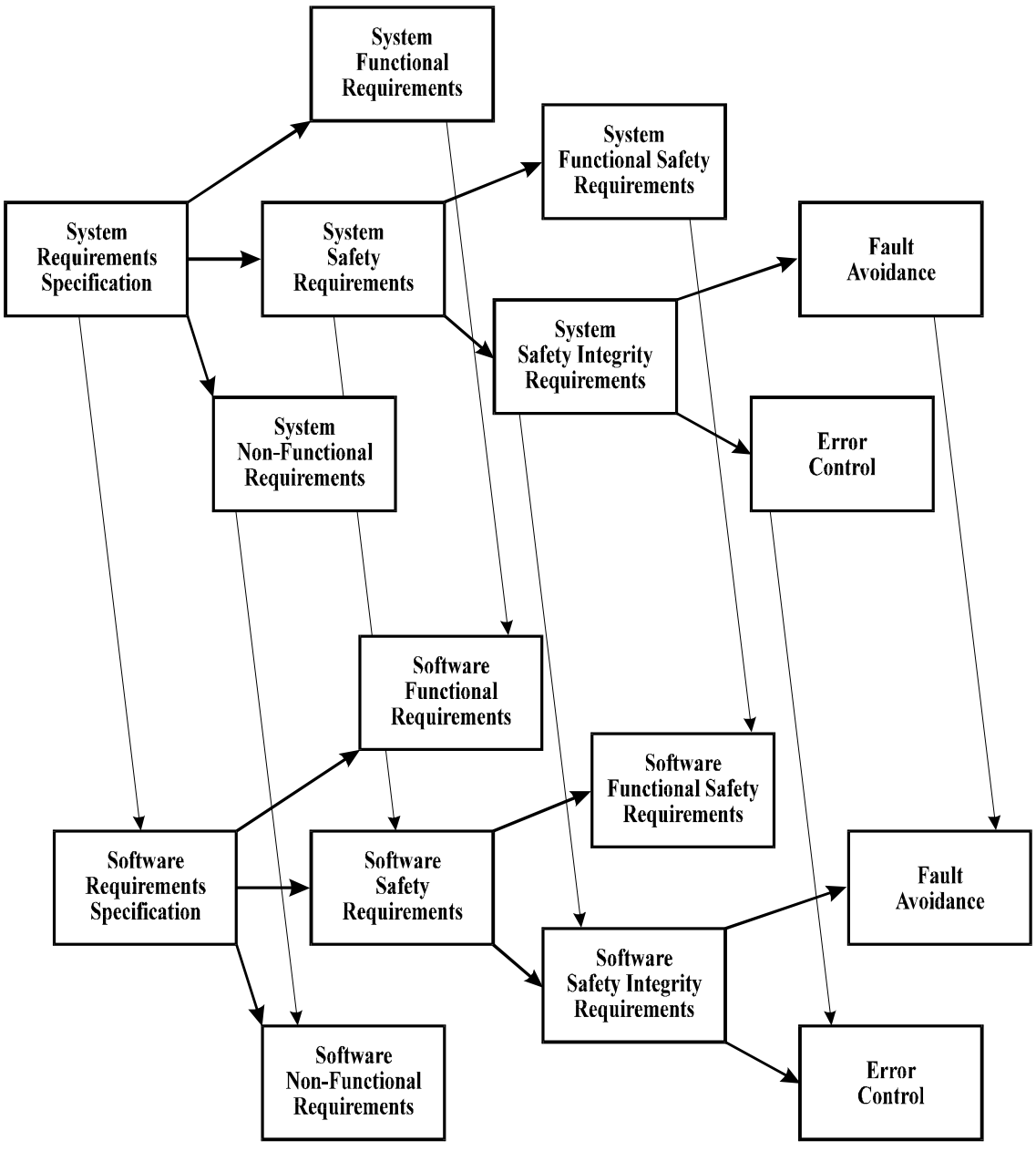
3.2.1 Statement of requirements

A statement of requirements is a broad outline description of the required system and is produced pre-development, during project initiation. It is sometimes referred to as a *User Requirements Specification* or features list, and serves to identify the functions required of the proposed equipment. More importantly, it also defines any safety-invariants (statements of the safety of the system that must apply in all circumstances) for the system. This may force the early introduction of design constraints (see Appendix C) identified as essential to a demonstration that the safety-invariants are not violated. Safety should be considered according to the basic functions required of the system, and what the current state-of-the-art for hardware and software will safely allow.

The safety-invariants for the system are identified by a Preliminary Hazard Analysis or similar (see Appendix D (see also [1])). This phase is essential, as it underpins all the safety-related aspects of the system development. A subset of the system safety-invariants will relate to the software (depending on the partitioning of the system functions between the hardware and the software) and will drive the software safety verification process. This information could be used as the specification for a safety monitor — a processor separate from the main control processor.

3.2.2 System requirements specification

A specification can be defined as a rigorous set of requirements that encapsulate the need that the system has to fulfil. A specification should be unambiguous, complete and not contain any unnecessary information which may cause confusion, i.e. be concise. The former can be facilitated by formalising the specification with a mathematical model which can be manipulated by mathematical techniques.



Note: Information Breakdown from Left to Right and from Top to Bottom

Figure 3.1 - Document Interrelationship 3

The property of completeness is by far the most difficult to achieve properly. Its definition is simple, a specification is complete if for all inputs a corresponding output is specified. Whilst this can be achieved in some situations, it may often be necessary to use another property that a specification must possess, namely that of authoritativeness. A specification is authoritative if it can be used to determine whether the system is correct without reference to a higher authority. Thus "this device will function in all *normal* circumstances" is a statement whose meaning may ultimately have to be decided by litigation if there is a disagreement as to what is meant by "normal", and is therefore not authoritative. A specification must therefore be the highest authority and normally takes precedence over contract terms.

A *System Requirements Specification* will usually contain details of the functions that the system is to perform, as well as certain mechanical, electrical, chemical, etc. properties that it should also have. It is important that these various properties should be readily identified.

3.2.3 Functional requirements analysis

Functional requirements analysis should be performed prior to generating the *Software Requirements Specification*. Its purpose is to ensure that the functional requirements for the project are clear and unambiguous; to form a formal basis for the generation of the *Software Requirements Specification*; to form the design document for the system, to ensure that the hardware and software are designed from the same set of rules.

The form of the functional requirements may be diagrammatic, or textual with defined semantics. A formal specification language (see Appendix E) may be used. The output should be initially analysed by a team, possibly using an animation tool (see Appendix H), whose responsibility is to identify any omissions from the requirements. After this it should be analysed, preferably by an automated checking tool, and any ambiguities, inconsistencies or errors should be identified and corrected.

The output from the functional requirements analysis is passed to the software design team, verification and validation team, software quality team.

3.3 Software requirements development process

Responsibility: *Software Manager and Software Engineer*

The objective of the software requirements development process is to produce a definition of what the software is to do. The *Software Requirements Specification* produced from the *System Requirements Specification* form the high-level requirements. Typically, these are further developed during the software design process, thus producing one or more successive levels of design material. The development of software architecture involves decisions made about the structure of the software. During the design process, the software architecture is defined and low-level requirements are developed, from which source code can be directly implemented without further information.

In some cases, source code may be generated from high-level requirements. In these cases, the high-level requirements should be treated as both high-level and low-level requirements for the purposes of this document.

Each level of development may also produce derived requirements, as a result of a design decision, which are not directly traceable to higher level requirements. The system safety assessment process should determine their effects on the safety requirements.

3.3.1 Software requirements development process activities

The inputs to the software requirements development process are the *System Requirements Specification* and the *Software Design Plan*. When planned transition criteria have been met, the process uses the *System Requirements Specification* to develop the high-level requirements.

The system's functional and interface requirements allocated to software should be analysed for clarity, consistency and completeness, to ensure that the *System Requirements Specification* is understood. Each system requirement allocated to software to preclude system hazards should be identified in the high-level requirements.

The high-level requirements should comply with the *Software Design Plan*, and should at a minimum be verifiable and consistent. They are based on the requirements allocated to the software and the system architecture. The requirements from these sources are developed during the software requirements development process into functional, performance, interface and safety-related requirements. The capabilities should be stated in quantitative terms with tolerances where applicable. Descriptive material and diagrammatic representation may be used as appropriate. The high-level requirements should not describe design or verification detail except for required design constraints (see Appendix C).

Each high-level requirement should be traceable to one or more requirement of the *System Requirements Specification*. Conversely, each system requirement allocated to software should be traceable to one or more high-level requirements. Derived requirements should be recorded.

Inadequate or incorrect inputs detected during the software requirements development process should be reported to the input source process for clarification or correction. This process is complete when its objectives, and the planned objectives of the associated integral processes are satisfied.

3.4 Software requirements specification

Responsibility: *Software Manager and Software Engineer*

The *Software Requirements Specification* for the software sub-system will be obtained from the *System Requirements Specification* of the whole system. For clarity it may be preferable

for separate documents to be drawn up relating specifically to the software sub-system. In addition to describing the software itself the documents should also indicate:

- the interface between the sub-system under consideration and any other sub-system, whether a direct connection exists or is planned
- the constraints between hardware and software
- any special operating conditions such as commissioning, repair and maintenance.

The *Software Requirements Specification* is made up of three distinct parts, the *Software Functional Requirements*, the *Software Safety Requirements* and the *Software Non-Functional Requirements*. When combined they form the *Software Requirements Specification*, which should be verified.

3.4.1 Software functional requirements

The *System Requirements Specification* is developed from both the operational requirements and the safety-related requirements which result from the system safety assessment process.

The systems requirements for in-vehicle software will usually specify two types of behaviour of the software:

a) The software shall perform specified functions that form the core of the *System Functional Requirements*.
b) The software shall NOT exhibit specific anomalous behaviour that could cause a hazard as identified by the system safety assessment process. Additions to the *System Requirements Specification* are generated to eliminate these anomalous behaviours.

These additions to the *System Requirements Specification* should then be transformed into additions to the high-level *Software Requirements Specification* that in turn force specific types of verification activities.

The *Software Requirements Specification* is solely concerned with the functions that the sub-system needs to carry out. It is not concerned with how they are to be carried out, nor with the degree of integrity required of the functions (i.e. the *Software Functional Requirements* is not concerned with the degree of certainty, or the likelihood, that a specific function will be carried out but rather establishing and defining those functions that are necessary for the (safe) operation of the sub-system (see also Section 3.4.3)).

When appropriate the *Software Functional Requirements* should be written using a formal specification language (see Appendix E) suitable for the sub-system. It should describe the desired functions of the sub-system in terms of what has to be done, not how it is to be done.

Sometimes the *Software Requirements Specification* will be describing an entirely new product whose implementation is unknown at this stage. However, the sub-system will often have to fit into an existing system, and so there may be a number of imposed constraints

which will also have to be stated. (Notes on the contents of the *Software Functional Requirements* can be found in Appendix F.)

3.4.2 Software non-functional requirements

The purpose of the *Software Non-Functional Requirements* is to define, for the software design process, the capabilities of the software and the constraints, including hardware, under which it must operate.

This data should contain, but not be limited to:

a) Allocation of system requirements to software, with attention to safety requirements and potential safety hazards.
b) Functional and operational requirements under all modes of operation.
c) Performance and test criteria (for example, precision, accuracy).
d) Hardware and software interfaces (for example, protocols).
e) Timing requirements and constraints.
f) Memory size constraints.
g) Failure detection and monitoring requirements
h) Software level(s).

3.4.3 Software safety functional and integrity requirements

The *Software Safety Requirements* are divided into the *Software Functional Safety Requirements* and the *Software Safety Integrity Requirements*.

In the *Software Functional Safety Requirements* all the safety functions are described. These functions must be carried out by the safety-related control system should a hazardous failure occur in the technical process or in the safety-related system itself.

In the *Software Safety Integrity Requirements* measures are specified to ensure that all safety functions are performed in order to meet the estimated integrity level. The measures are divided into measures to avoid, and measures to control systematic and random failures. The *Software Safety Integrity Requirements* establishes the requirements for achieving the level of safety integrity and dependability necessary to perform the functions specified in accordance with Section 3.4.1. Whilst the system wide plan for achieving safety will be found in the *System Safety Integrity Requirements*, there are in addition many features that should be contained within a programmable sub-system to maintain its own integrity. (Notes on the contents of the *Software Safety Integrity Requirements* can be found in Appendix G.)

3.5 Requirements analysis

Responsibility: *Software Manager and Software Engineer*

Analyses should be conducted as described in Section 2.5.

3.5.1 Software requirements analysis

Software requirements analysis should be performed prior to generating the software architecture design. Its purpose is to ensure that the software performs the functions defined in the *Software Requirements Specification*; to identify ambiguities, inconsistencies and errors in the *Software Requirements Specification*.

The form of the *Software Requirements Specification* may be diagrammatic, or textual with defined semantics. A formal specification language (see Appendix E) may be used. The output should be initially analysed by a team, possibly using an animation tool (see Appendix H), whose responsibility is to identify any omissions from the requirements. After this it should be analysed, preferably by an automated checking tool, and any ambiguities, inconsistencies or errors should be identified and corrected.

The output from the software requirements analysis is passed to the software design team, software quality team and verification and validation team.

3.5.2 Software design review

The objective of a software design review is to detect and report requirements errors that may have been introduced during the software requirements development process and the software design process.

These reviews and analyses should confirm that the full *Software Requirements Specification* is accurate, complete and can be verified. The high-level requirements should meet the *System Requirements Specification*, and the low-level requirements should meet the high-level requirements.

The review topics should include:

a) Compliance: For high-level requirements, the objective is to ensure that the system functions to be performed by the software are completely defined, that the performance and safety requirements have been correctly reflected in the high-level requirements, and that justification is provided for derived requirements. For low-level requirements, the objective is to ensure that the low-level requirements correctly reflect the high-level requirements, and that derived requirements resulting from design decisions are correctly defined and justified.

b) Accuracy and consistency: For high-level requirements, the objective is to ensure that each high-level requirement is accurate, unambiguous and sufficiently detailed, and that the requirements do not conflict with each other. For low-level requirements, the objective is to ensure that each low-level requirement is accurate and unambiguous, and that the low-level requirements do not conflict with each other. These objectives could be met using methods supported by formal specification techniques.

c) Compatibility with the target hardware: The objective is to ensure that no conflicts exist between the high-level requirements and the hardware features of the target system. Special attention should be given to the use of resources (e.g. bus loading),

system response times and the input-output hardware.

d) Verifiability: The objective is to ensure that each requirement can be verified. If not, the requirement is inadequate and should be revised.

e) Conformance to standards: The objective is to ensure that the requirements standards have been followed during the software requirements development process and that any deviations are justified.

f) Traceability: For high-level requirements, the objective is to ensure that the functional, safety and performance requirements of the system which are allocated to software, are completely developed into the high-level requirements. For low-level requirements, the objective is to ensure that the high-level requirements and the derived requirements have been developed into the low-level requirements for the software.

g) Algorithm aspects: The objective is detailed examination of the proposed algorithms, giving special attention to confirmation of accuracy and to the behaviour in the regions of discontinuities.

3.5.3 Checklists

The use of a checklist is important in examining software specifications for common errors [6]. Adoption of a checklist will provide a stimulus to critical appraisal of all aspects of the system rather than to lay down specific requirements. The general format is as a set of questions to be completed by the person performing the checklist. To accommodate wide variations in software and systems being validated, most checklists contain questions that are applicable to many types of system. In this case they must be interpreted as seems most appropriate to the particular system being assessed. Equally there may be a need, for a particular system, to supplement the standard checklist with questions specifically directed at the system under development.

In any case it should be clear that the use of checklists depends critically on the expertise and judgement of the engineer selecting and applying the checklist. As a result the decisions taken by the engineer, with regard to the checklist(s) selected, and any additional or superfluous questions, should be fully documented and justified. The objective is to ensure that the application of the checklists can be reviewed and that the same results will be achieved unless different criteria are used.

The object in completing a checklist is to be as concise as possible. When extensive justification is necessary this should be done by reference to additional documentation. Pass, Fail and Inconclusive, or some similar restricted set of tokens should be used to record the results for each question. This conciseness greatly simplifies the process of reaching an overall conclusion as to the results of the checklist assessment.

Questions to be asked include but are not restricted to (see [7]):

1) Have all inputs to the system been defined?
2) Have their sources been identified? (human agent, other machine, communication lines, peripheral types, etc.).

3) Have their types been specified? (Analogue, digital, electrical, acoustical, optical, etc.).
4) Have the range, scaling, format, byte layout, etc. been specified?
5) Have validity checks been specified?
6) Have all the accuracy levels been defined?
7) Have all outputs from the system been defined?
8) Have all aspects of system performance been defined?
9) What is the throughput of the system under different loads?
10) What are the response times in different circumstances?
11) What must be the system's response to failures of software, hardware, power, etc.?

Wherever possible, checklist questions must be designed to elicit definitive answers: "is the output of this routine between X and Y" rather than "does this routine do what it is supposed to?" A wide range of questions that may be asked is described in [6].

4. Design

4.1 Partitioning

Responsibility: *Project Engineer and Software Manager*

Partitioning refers to the technique of providing isolation between software components to limit the scope of the verification process. If protection by means of partitioning is provided, the software level for each partitioned component may be determined using the most severe failure condition category associated with that component.

Potential breaches of the partitioning which should be considered when designing the protections are:

a) Hardware resources: processor, memory, I/O devices, interrupts, timers.
b) Control coupling: vulnerability to external access.
c) Data coupling: shared or overlaying data (including stacks and processor registers).
d) Failure modes of all hardware devices associated with the protection mechanisms.

These aspects should be considered by the development and the verification processes. This should include the degree and the methods of partitioning. The degree refers to the degree of interaction of the partitioned components. The methods refer to whether the protection is implemented by hardware or by a combination of hardware and software. If the partitioning protection involves software, then the software should be assigned at least the highest level of the partitioned software components.

The *Software Non-Functional Requirements* should include any partitioning requirements allocated to software as well as how the individual software components interact with each other.

The *Software Verification Specification* should describe the reviews, analyses and tests for confirming that partitioning operates as required.

4.2 Maintenance of consistent design data

Responsibility: *Software Manager and Software Engineer*

It is essential that all data used to analyse any aspect of the system during its design life-cycle remains consistent and traceable to the *Software Requirements Specification*. It is often the case that, because of the acknowledged flexibility of software-based systems, and the ease with which they may be modified, requirements may change during the design cycle. This can result in incorrect conclusions being drawn during the later stages of the life-cycle: it is possible for data used to test some modules to be incorrect for other modules, or for the composite program.

It is inevitable that some changes to the requirements will be necessary as the project progresses. When change is to be effected, it should be the subject of a review by a team representative of all aspects of the project, and procedures employed to ensure that all relevant areas' specifications are updated, back to the *Software Requirements Specification*, and, if necessary, the *System Requirements Specification*. Traceability must be maintained.

If the result of change is that the data on which analysis, test, and review are based is changed, it may invalidate earlier stages of the analysis and review. Consideration must be given to ensuring that the results of all these procedures remain valid, even if it requires repeating a major part of earlier effort.

It is usual for data about software projects to be held in a database to enable automated means of version and modification control, etc. There should be validation procedures within the database to ensure that changes to data maintain consistency within defined system constraints. There are references that may be consulted on maintaining database integrity, notably [8].

4.3 Design and development reviews

Responsibility: *Software Engineer*

In practice any milestone review will be split into a number of different reviews with very specific aims. The degree of independence of the review team can also vary from a programmer checking his or her own code, to a committee that contains no member of the team which has performed the work. Reviews should be performed as part of a *Quality Assurance Plan*, with any changes that are recommended as a result of the review being performed under a suitable configuration management system (see ST2 Integrity).

4.3.1 Design reviews

The purpose of design reviews is to establish that the design process is producing the required output at each relevant stage. Design reviews may take place at any defined milestone in the project, in order to review the procedures and results of work prior to that point. They may also be used as authorization tools for the next work phases.
It is not the case that all reviews are concerned with the technical issues of software creation: there will also be cost, planning, and general quality reviews as part of the project quality assurance process.

Review teams should not be too large: three or four people should be assigned as principal reviewers. Any others who have a contribution to make may be co-opted as necessary to the team, or else comments may be invited from a wider audience, to be reviewed at the actual review meeting [9].

4.3.1.1 Software design review

Software design reviews are multi-functional and are intended for defect detection, education and overall strategy decisions. They are a quality assurance mechanism which involves a group of people examining part or all of a software design with the aim of finding anomalies in that design. The conclusions of the review are formally recorded, in the *Software Design Review Results* and passed to the author or whoever is responsible for correcting design faults. The design review should be done before any programming commences.

The completion of the verification of the *Software Module Specification* is an important milestone in the development of the software system. This milestone signals the end of the software design and enables the software programming to begin.
[9] identifies three activities referred to under the general idiom of design reviews:

- *Design or program inspections* — these are intended to detect detailed errors in the design and to check whether or not standards have been adhered to. It is common practice for reviewers to make use of a checklist of possible errors and to use this checklist to drive the review process. This technique can be equally well applied to detailed designs as to programs.
- *Quality review* — The design work of an individual or of a team is reviewed by a panel made up of project members and technical management. This type of review is distinct from a design inspection in that the design may not be described (or even developed) in detail. The review process is intended to detect serious design faults and mismatches between the design and the specification.
- *Management product reviews* — This type of review is intended to provide information for management about the progress of the software design project. It is not intended for detailed design validation. Although the design is discussed, the main concerns of this type of review are design costs, plans and schedules. This type of review is an important project checkpoint where major decisions about the readiness of the project for implementation or even product viability are made. It is a management activity and is not part of the design validation process.

The software design review, which should finish before the programming begins, should be conducted by an independent team, which may include some of the design team, and must include some of the quality team.

The design review committee should review each design description. The review should consider the correctness, completeness, consistency, absence of ambiguity, general style and quality of the document, and its conformance to the *Quality Assurance Plan*. It should establish that it correctly implements the higher level design descriptions or the *Software Requirements Specification*, taking into account the formal arguments produced by the design team. The committee should first produce a *Software Design Review Plan*, which should address the following:

- the adequacy of the *Software Requirements Specification* to meet the requirements set out for the *Quality Assurance Plan* for safety, and in the *System Requirements Specification* for functionality, reliability, performance and safety,
- the adequacy of the *Software Detailed Design Specification* to meet the *Software Requirements Specification* with respect to consistency and completeness down to and including the module level,
- the decomposition of the *Software Detailed Design Specification* into the *Software Module Specification* with reference to:
  - feasibility of the performance required,
  - testability for further verification,
  - readability by the development and verification team,
  - maintainability to permit further evolution.
- the adequacy of the *Software Verification Specification*, and the results of its application so far.

Other related design material:

- the *Software Integration Test Results*.

4.3.1.2 Hazard analysis

Formal reviews of the hazard analysis and safety risk assessment should be carried out as part of the project design reviews. These reviews should include the controllability categories identified and the safety integrity levels assigned to functions and components (see ST2 Integrity); and the results of the analyses carried out. They should endorse the associated entries in the *Hazard Log* and supporting documentation.

4.3.2 Progress reviews

The objective here is to detect and report errors that may have been introduced during the development of the software architecture. These reviews should confirm that the process products are accurate, complete and can be verified or validated. The topics should include:

a) Compliance with the high-level requirements: The objective is to ensure that the software architecture is compatible with the high-level requirements. Special attention should be given to functions that are required to maintain system integrity, (for example, partitioning schemes) and that justification is provided for any portion of the software architecture that is not directly traceable to the high-level requirements.

b) Accuracy and consistency: The objective is to ensure that a correct relationship exists between the components of the software architecture (data flow and control flow).

c) Compatibility with the target hardware: The objective is to ensure that no conflicts exist between the software architecture and the hardware features of the target system (e.g. issues related to initialisation, asynchronous operation synchronisation and interrupt levels).

d) Ability to be verified and/or validated: The objective is to ensure that the proposed implementation can be verified (e.g., no unbounded recursive algorithms), and/or

validated.

e) Conformance to standards: The objective is to ensure that the standards have been followed during the software design process and that deviations are justified. Special attention should be given to complexity restrictions and to the use of any design construct that could compromise safety.

f) Traceability: The objective is to ensure that all the requirements are traceable to the software architecture.

g) Partitioning integrity: The objective is to ensure, if applicable, that the means through which a partition may be breached are prevented, isolated or eliminated.

4.4 Design material

Responsibility: *Software Engineer*

4.4.1 Software design plan

Once an integrity level has been given to a sub-system that contains software, a plan must be made as to how this integrity level will be achieved. In particular, decisions must be taken as to:

- what extent formal methods (see Appendix I) will be used for the *Software Requirements Specification*,
- whether any special software architecture and development techniques are to be used to develop the software (see Section 5.2.1),
- the level of testing that will be performed.

The result should be the *Software Design Plan*, which should be confirmed by a review that it is currently acceptable, before proceeding further with the development.

Other related design material:

- the *Software Integration Plan*.

4.4.2 Software verification specification

Verification is concerned with the confirmation, or otherwise, of the self consistency of the various stages in the design and implementation phases. Each stage of the software development life-cycle should have a *Software Verification Specification* established for it. The plan should document all the criteria, the techniques and tools to be utilised in the verification process for that phase. It should describe the activities to be performed to evaluate each item of software for each phase to show whether the system functional, reliability, performance and safety requirements are met in accordance with the *Software Design Plan*. The level of detail should be such that an independent group can execute the *Software Verification Specification* and reach an objective judgement as to whether or not the software is adequate. The degree of independence required will depend upon the integrity level.

The result should be the *Software Verification Specification*.

4.4.3 Software detailed design specification

When performing the initial design of the software the following points should be considered. The software should:

- be minimum in size and complexity
- have features that facilitate:
  - abstraction, modularity and other features that control complexity
  - the clear and precise expression of:
    - functionality
    - information flow between components
    - sequencing and time related information
    - concurrency
    - data structures and properties
  - human factors (see [1])
  - verification and validation
  - software maintenance (e.g. information hiding, encapsulation)
- include self monitoring of control flow and data movement, and on failure detection appropriate actions should be taken
- be based on a decomposition into modules.

The result should be the *Software Detailed Design Specification* which, once it has been accepted should only be modified under appropriate change controls (see Section 2.10).

Other related design material:

- the *Software Requirements Specification*.

4.4.4 Software module specification

The program structure should be based on a decomposition into modules. Each module should have its own *Software Module Specification*, *Software Verification Specification* and *Test Plan* section of the *Software Validation Plan*. If the program is to be of any size (i.e. there are more than a few modules) then each module should have its own set of documentation. There should also be a *Software Verification Specification* and the *Test Plan* section of the *Software Validation Plan* for when the modules are put together (see Section 6.4.3):

- each module shall be assigned its own integrity level which will normally be that of the entire program
- each module should be understandable on the basis of its interface
- a top-down approach should normally be taken to the design of software except to certain parts, such as device drivers, where a bottom-up approach will be acceptable
- a module should be chosen and written with a view to easy testing
- where standard software from a manufacturer or supplier is used, it must be

shown to be acceptable for its proposed new use, usually by having been developed under an appropriate standard and in accordance with the MISRA Guidelines

- existing proven components should be used where possible (see Section 2.8.2).

The result should be a number of entries in the *Software Module Specification*.

5. Programming

5.1 Language qualities

Responsibility: *Software Manager*

The following are normally considered to be good attributes of a programming language (see also [1]):

- the language should be fully and unambiguously defined
- the language should be user or problem orientated rather than machine orientated
- widely used languages, or their subsets, are preferred to special purpose languages.

For safety-related software some procurers may insist on the use of a specific language or a predefined subset of a language (see Section 5.1.2.1), whilst others may allow negotiations to take place between the interested parties. The selection criteria for the languages should include the following:

- a formally-defined syntax
- a means of enforcing the use of any subset employed
- a well-understood, formally defined, semantics and a formal means of relating the code to the formal design
- block structured
- strongly typed
- translation time checking
- run-time type and array bound checking
- parameter checking
- conformance to an international standard and the use of a validated compiler for the host and target
- maturity
- extensive tool support, and tools that are trusted/validated
- exception handling.

The language should encourage:

- the use of small and manageable modules
- restriction of access to data in defined modules (*information hiding*)
- definition of variable sub-ranges
- any other type of error limiting constructs.

It may be necessary to prohibit the use of certain processor instructions or features of a language in safety-related software. This could be because of the:

- complexity of the instruction
- difficulty in predicting the effects of a failure
- difficultly in testing or simulating its operation
- nature of the use of mathematical formal methods (see Appendix I) in the specification and design
- requirement to use static analysis (see Appendix J)
- ambiguities and insecurities
- implementation independence.

Compilers/translators are used to convert high-level languages (source code) into a machine readable form (object code). However since the high-level code is expected to be portable between different computers (and possibly different operating systems) there will be a range of different compilers/translators for any given high-level language. Because of the important consequences of reliance upon a compiler/translator, especially if at a later stage in its life the safety-related code may use a different version of the compiler/translator, the designers and programmers should consider avoiding language features that are covered by the following list:

- *Unspecified behaviour* — this covers any source code language construct which will be legitimately compiled but for which the object code output produced by the compiler is not documented in the compiler semantics.
- *Undefined behaviour* — this covers source code the nature of which means that the executed results of the object code will be unknown e.g. run-time array overflow.
- *Implementation specific behaviour* — certain documented compiler/translator functions may be left to the installer as to the choice of implementation techniques. This may effect the object code behaviour. This will normally be consistent at a particular site but may differ between separate sites.
- *Local specific behaviour* — due to differing machine constraints the performance of the object code may differ on different machines/operating systems, even if these are at the same site.
- *Accidentally undefined behaviour* — unlike the *undefined behaviour* (above) the language features which are included in this category have been imprecisely defined unintentionally.
- *Empirically determined misbehaviour* — the compilation/translation of this construct is fully documented but is not what it was intended to be (e.g. in C the construct *a == b*).
- *Unexpected behaviour* — the behaviour of this language construct is not as would normally be expected, although it will be clearly documented as it has been defined (e.g. in C the code segment *if (fork {}) {proc1;} else {proc2;}*).

Most of these errors/features are well known throughout the programming community and are invariably documented with the compiler. The existence of such a known error/feature list is only useful whilst the compiler is not altered in any way.

If a feature which exhibits any of the characteristics in the previous lists, is relied upon in

meeting the safety requirements, then the code will need to be checked, preferably by the use of a well tried or certified tool, to ensure that the characteristics have been avoided.

5.1.1 Guidelines for choosing languages and translators

The following are guidelines for the selection of a programming language, including the operating system, and its associated translator or compiler for use in a safety-related software system. Additional advice can be found in [10] and EWICS [11], from which some of the points below are taken. [12] reports on the suitability of specific languages for safety-critical applications.

1) Logical Soundness: The programming language should be logically coherent and unambiguous with formally defined semantics, in order that reasoning about programs is possible.
2) Lack of Complexity: A formal definition of a language is necessary to underpin the informal everyday description. The formal definition then forms a basis for a verified compiler and for other language processing tools, such as formal verification tools. However if the language is too complex (the formal definition of Ada's runs to eight volumes) then its logical coherence becomes impossible to establish, precluding the social process essential to its justification [13], reasoning about programs becomes uncertain, and formally-verified compilers for such a language will also not be possible.
3) Expressive Power: An expressive language, with structured statements, strong typing, and procedural and data abstraction, transfers the work of producing correct programs from the programmer and verifier, to the language and compiler. An expressive language eases refinement from specification to implementation.
4) Security: An insecurity is a feature of a programming language whose implementation makes it very difficult (or impossible) to detect violation of the language rules other than by execution. All standard programming languages suffer from insecurities. If any insecurities in a program pass undetected, analysis of the program, formal or otherwise, will be invalidated.
5) Verifiability: We should ideally be able to show through formal proof that a program is an implementation of its specification. This requires a relatively simple formal definition of the language. The language should adapt naturally to refinement (i.e. the top down creation of a program from its specification). In addition, to limit the complexity of the task, we need to be able to reason about fragments of code out of context of the entire program and to map these with fragments of specifications. This requires support for procedural and data abstraction, as well as suitable scope and visibility rules.
6) Bounded Space and Time Requirements: In real-time control applications we must ensure that adequate memory is available for the running program. Dynamic storage allocation is therefore to be avoided where possible and all constraints should be statically determinable. Proof methods should be used to obtain finite bounds on the number of loop iterations in order to satisfy execution times.
7) Compilers with a current national or international certificate showing conformance with the appropriate language definition should be used. Compilers without such a

certificate may be used provided that they have been thoroughly tested. (The adequacy of such testing must be justified.)

8) A trusted compiler for which there is a long history of use, and for which all faults have been fully recorded, may be considered provided it can be shown that all these known faults have been circumnavigated. The record of known faults is only valid provided the compiler undergoes no modification.

9) Problem oriented languages are strongly preferred to machine oriented languages. The language used should:
   - encourage the use of small and manageable modules
   - allow access to certain data to be restricted to defined modules
   - allow the definition of variable type sub-ranges.

   However the use of an assembly language may be necessary in the following circumstances:
   - memory constraints
   - only assembly language available
   - performance constraints
   - customer constraints.

10) Programming standards should be used for all safety-related software. These programming standards should specify good programming practice and should proscribe unsafe language features, which are difficult to verify, such as:
   - recursion should be simple to a known depth — though a recursive solution will be much clearer than an iterative one
   - pointers, heaps or other dynamic memory usage (avoid if possible)
   - floating point arithmetic (avoid if possible)
   - interrupt handling at source code level — though a system with synchronous interrupts may be less complex than one without
   - undeclared variables
   - implicit variable initialisation or declaration, assumptions regarding variables which have not been initialised
   - complex language constructs such as variant records, functional parameters and equivalence
   - multiprocessing on a single processor
   - data not generally entered by the user at run-time
   - object code patching
   - unconditional jumps excluding subroutine calls
   - multiple entries or exits of loops, blocks or subprograms
   - procedural parameters.

Notice that this widely accepted list of unsafe language features is in fact identifying features for which current techniques cannot guarantee verifiable behaviour. However there is an emerging belief that it should be the underlying problems that should be banned and not the consequences. For example, recursion is difficult to verify because it may cause the system to run out of memory unpredictably. If technology advances so that recursion can be assured of bounded memory usage then this technique should no longer be listed. Thus it is "running out of memory" that should be avoided, and consequently any program technique that exhibits this behaviour.

Sometimes the use of one or more of the above features may be unavoidable, but the predictability of behaviour remains the concern and an early discussion with the assessor (see [1]) should take place to agree the verification and validation procedures that should be used.

11) A programming language used for critical software should not include multi-tasking, multiple use of operators, etc. A subset of a language that does not use these functions would be allowed.
12) Where available automatic testing tools should be used, for example debuggers.
13) Where available integrated development tools should be used. These include structural analysis and design tools.
14) The behaviour of the code should be predictable such that it can be modelled statically and analysed through logical argument.
15) Only the simplest memory management techniques should be used in safety-related software. Static, compile time allocation should be used, to reduce the risk of a program failing at run time. A simple stack, for subroutine entry and exit, is permissible, provided that static analysis checks (see Appendix J) ensure that the entry and exit behaviour is correct.

As might be expected, no standard languages score highly in all of the above. Examples of common difficulties are:

- *subprogram side-effects* — whereby variables in the calling environment may be unexpectedly changed, presenting further problems when this occurs in function evaluations within an expression.
- *aliasing* — whereby two or more distinct names refer (possibly inadvertently) to the same storage location, so that a change to one variable may also modify a seemingly different variable.
- *failures to initialise* — resulting in the use or potential use of a variable before it has been assigned a value.
- *expression evaluation errors* — which might result, for instance, from an array subscript being outside the permitted index range, from the use of a partial function with invalid arguments (e.g. division by zero), or from arithmetic overflow.

5.1.2 Recommended languages

It is desirable that the language is supported by a suitable translator (see ST2 Integrity), appropriate libraries of pre-existing modules, a debugger and tools for both version control and development.

Low level languages, in particular assembly languages, present problems due to their machine oriented nature.

Ada is extremely complex and contains many insecurities [14]; Pascal's support for abstraction is limited; C is poorly defined, with many insecurities, and is generally regarded as unsuitable for use in high-integrity applications [12]; Modula-2 (perhaps the most suitable

language) suffers from a lack of tool support (and users!); assembly languages are very informally defined, have limited expressive power and are too permissive (this prompted an early draft of DEFSTAN 00-55 to ban the use of assembly languages, although it is now accepted that this is unrealistic). Informal comparative studies can be found in [12].

An approach might be to define a new language, which satisfies the requirements. In fact, a few languages have been developed with this objective, including Euclid [15], Newspeak [16] and m-Verdi [17], although none have become established (and some of them in any case do not meet all the requirements). The reluctance of the industrial computing community to adopt a new language is largely because of the enormous investment required to develop reliable compilers and other support tools, and to train and establish a community of users sufficient for its use in critical systems. (The political and financial support behind Ada, for example, was very great.)

5.1.2.1 Safe-subsets or restricted subsets

Safe-subsets of languages are now being presented which avoid constructs that make program analysis difficult. It is the belief amongst the community that such subsets really deserve the title *safer-subset*. In fact many languages which have insecure constructs are accompanied by a large range of supporting tools which can help to eliminate the standard problems. In this way even *unsafe* languages may be considered safe.

Examples of subsets that have been specifically created for safety-critical applications include SPARK (one of three Ada subsets) [18], SPADE-Pascal [19], and a M68020 assembly language subset [20]. Each "high-integrity" subset is developed from the parent language by disallowing any construct which is in some sense unverifiable (see Section 5.1.1), and by adding annotations (formal comments) to help eliminate insecurities from those constructs which remain, for example by defining the import-export relations for the module (as given in its detailed design description). This allows us to build on existing technology by providing more rigorous definitions of commonly used languages, by training users in programming techniques which assist in the analysis process, and by providing analysis tools.

5.2 Practices

Responsibility: *Software Engineer*

The programming phase for safety-related software is characterised by discipline and "good practices" which contribute to making the code readily understood. For example:

- simple, single entry, single exit
- initialisation of data
- simple algorithms
- templates
- program description language for comments.

Inclusion (in the code) of assertions derived from formal specifications is recommended. Semantic and compliance analysis (which form part of static analysis (see Appendix J)) should be performed for safety-related software and this requirement will have a dominant effect on the whole development process.

Tools to check and analyse software are now available and their use in detecting incorrect code, and code which is non-compliant to programming rules make an increasingly important contribution to the quality and reliability of the software.

Where well tried code already exists, this should be used in preference to less qualified code (see Section 8.6.4).

Programming standards should be used for all safety-related software. These standards shall specify good programming practise and shall proscribe unsafe and ambiguous language features. Such restrictions should, where possible, be enforced by automatic means.

Where possible compilers or translators that have a "Certificate of Validation" to a recognised National or International standard should be used. Where possible integrated development tools, and automatic debugging tools should be used.

5.2.1 Currently accepted good software engineering practices

The principles detailed here are intended to enable software to be produced which contains a minimum number of faults and should be readily verifiable.

Additional advice can be found in the EWICS guidelines to the development of critical software [11], which has been adopted, almost verbatim, by the International Electrotechnical Commission (IEC). A description of many methods for the design of safety-related software can also be found in the current draft of IEC/SC65A/WG9.

1) The software design should include the self-monitoring of control flow and data movements. On error detection appropriate action should be taken (defensive programming).
2) The program structure should be based on a decomposition into modules. Each module should have its own *Software Module Specification* and *Software Module Test Specification*.
3) Each module should be readable and comprehensible from start to finish. Its function should be understandable on the basis of its interface.
4) Good documentation must be provided.
5) Methods of obtaining the required reliability including self-monitoring should be chosen at the outset of the development.
6) A top down approach to software development is preferred.
7) A conceptual model of the software system structure should be adopted at the beginning of each software project and only modified under appropriate change controls.
8) Each program and module should be written with the aim of allowing easy

verification.

9) Where standard software from a third party is used, it must be assessed to the same level as for any newly developed software (see [1]).
10) Existing verified components should be re-used where possible.
11) Strong component cohesion should be adopted to avoid unrelated concepts being grouped together. [21] identifies several levels of cohesion.
12) Where possible components should be loosely coupled and shared variables avoided.

5.2.1.1 Object orientation

Currently, it is not clear whether object oriented languages are to be preferred to conventional ones. See Appendix K for a discussion of issues such as available techniques, advantages and limitation.

5.2.1.2 N-version programming

N-Version programming is sometimes suggested as a means of providing software diversity when developing a voting system. The principal here assumes that two, or more, diverse programs are not likely to have faults in the same place. The problem is how to create the diversity. The original method of just giving the same specification to independent programming teams has been shown not to produce the diversity required [22]. The reason for this effect is primarily due to the similarity of education and training of all the personnel involved. This can result in similar approaches being taken by each of the teams even when different programming languages are used. This in turn will cause the difficult sections of code to occur in the same place in each program. Now since faults will tend to occur in the difficult sections of code such a form of diversity may not be as reliable as was supposed. In order to achieve real diversity a number of other approaches have been proposed. One is to use a different system model for each program (e.g. finite state machines, Petri nets, etc.) in the hope that the difficult sections will be at different places in the logic. Another is to use completely different programming techniques (e.g. functional programming, neural nets (see Appendix L), etc.). A possible limitation of these ideas however, is that the computation times can be very different between the various versions produced, and thus this approach may not be suitable in a real-time system.

5.3 Comments

Responsibility: *Project Engineer and Software Manager*

Commenting is an integral part of any well prepared program and is vital for code validation. Comments should be added as the code is being written and should describe the nature of any functions, identify problems together with their solution. If code has been poorly commented then maintenance becomes much harder, even if this is performed by the original programmer. There are three commonly accepted levels of commenting:

- *module* — typically in the form of a header providing version number, modification history, authorship, etc.

- *procedure/function* — recording the *Software Requirements Specification* together with entry and exit conditions, describe usage and parameters, show how variables relate to each other, etc.
- *code* — these typically introduce new variables, explain algorithms, identify the destination of jumps, describe loop invariants, etc.

5.4 Modularity

Responsibility: *Software Engineer*

Modularity was the first fault-avoidance technique to be widely used. The system is broken down into a number of modules that provide a concise interface to a function. "Information hiding" and "defensive programming" techniques can then be used. In this way any code using a given module need only to have information about the interface, and if this module were replaced by one that had a fault corrected or had higher performance, the rest of the modules in the system would not have to be redesigned. Furthermore, the module could perform sanity checks on the parameters passed to it in order to verify that it is being asked to perform an operation within its scope. This style of defensive programming is analogous to defensive driving in that the module can anticipate errors, instead of waiting for them to occur.

5.5 Compiler conformance

Responsibility: *Software Manager*

Programming languages cannot be considered in isolation from the compilation support provided for them. Compilers are complex programs which may themselves introduce faults. Some argue for the use of assembly code in small, very high-integrity systems simply to avoid this problem (assemblers are much simpler and are therefore less error prone). Clearly, compilers of the highest integrity are needed.

Currently, no formally-developed compilers are commercially available ([23] documents the verification of a Pascal-like compiler). The construction of such a tool is extremely difficult, and for standard languages such as Ada and Pascal, it is perhaps beyond the state of the art to produce a formally verified, efficient compiler. However, academic interest in this subject remains, and progress can be expected. In the meantime we must rely on less rigorous development techniques, and certification of compilers by recognised authorities, such as, in the UK, the British Standards Institution (BSI) (e.g. Pascal), and the National Computing Centre (NCC) (e.g. Ada). Some work on the verification of compiler-generated code has also been carried out.

Although the suite of tests used by the compiler validation centres cannot be relied upon to establish the "correctness" of the object code generated in all circumstances, it does at least help to demonstrate that the compiler conforms to the language standard - a minimum

requirement. It should also be noted that compiler "evaluation" centres also exist (e.g. BSI). Evaluation provides further assessment of the compiler, more generally relating to its fitness for purpose. The evaluation software or reports on evaluated compilers can be purchased.
It is often argued that the optimisations performed by a compiler are complex and hence should not be used for high-integrity software. However, work into the verification of compiler generated object code (from Ada to M68020) has shown that the situation is not as clear cut. It was found that for the analysis to proceed with the minimum of difficulty it was often necessary to use the compiler with optimisation enabled. As the compiler generates intermediate code assuming optimisation is enabled, very often the unoptimised code contained a significant number of additional program paths and redundant statements. Optimisations which reduced this redundancy were helpful. However, optimisations which cause "code motion" - hoisting loop invariants, reduction of loops to a common form, were unhelpful. In the former case the correspondence between an execution state in the source and in the target program had not been clearly preserved (making it difficult to re-use proof information from the source program), and in the latter case because the optimisation increased the complexity of the code fragment to be verified. Clearly, greater control over the optimisations performed by existing compilers would be helpful.

In summary compilers should meet the following requirements:

- they should be validated with an approved international or national validation certificate. It is necessary to ensure that the certificate does indeed relate to the current version and use intended in the project
- they should be verified to the extent indicated by the hazard analysis and safety risk assessment ([1]). As a minimum they should have been developed within a recognised Quality Control System (e.g. AQAP 13 or ISO 9000)
- if the object code is not shown by formal arguments to be equivalent to the source code, the compiler should be classified as safety-critical and should have been developed to the requirements of the MISRA Guidelines.

5.6 CASE tools

Responsibility: *Project Manager and Software Manager*

Computer-Aided Software Engineering (CASE) tools provide automated features to support software engineers, and can lead to improvements in productivity. Software tools to support programming have been available for some time, but the cheaper tools now available for personal computers means that they can now be made available to many more software developers than was previously possible.

There are now a great variety of other CASE tools to assist the process of software development. Along with interactive editors and debugging tools, the large amounts of backing store available means that data-base programs to keep track of code and documentation can be developed. The value of such tools is now further increased, beyond their possible use on timesharing systems which only offered a restricted amount of

computational power to each user, with the introduction of powerful, networked single-user workstations.

Individual software tools, used in conjunction, are very valuable but the true power of a CASE toolset can only be realised when these tools are integrated into a common framework or environment. Thus when a CASE tool uses its own specific intermediary language it should come together with a translator from a wide range of standard languages. It should ideally provide a range of static analysers which will test both the language syntax and its semantics. In addition a proof checker may be incorporated to enable the design to be verified against its requirements. In an integrated environment, any one tool can access the data produced by other tools. This allows both serendipitous tool combinations and incremental toolsets where operations common to a number of tools are implemented once and provided as a single tool.

The use of software tools can be beneficial for safety insofar as they can enforce compliance with standards and incorporate automatic checks which are performed systematically and improved with use. However assessment of a tool is required when recommended processes are eliminated, reduced, or automated by the use of a software tool whose output is not verified. The objective of the tool assessment process is to ensure that the tool provides a safety assurance at least equivalent to that of the process(es) eliminated, reduced or automated. If partitioning of tool functions can be demonstrated, only those functions that are used to eliminate, reduce or automate processes recommended in the MISRA Guidelines, and whose outputs are not verified, need to be assessed. If a tool is only assessed for use on a certain specified systems then its use for other systems may require further assessment.
Software tools can be classified as two types:

- *Software production tools* — whose output is part of the actual code and thus can introduce faults (e.g., code generators, some CASE tools). For example, a tool which generates source code directly from low-level requirements would have to be assessed if the generated source code is not verified.
- *Software verification tools* — which cannot introduce faults, but may fail to detect them (e.g., type checkers, analysis or test tools). For example, a static analyser, which automates a verification process, should be assessed if the function that it performs is not verified by another activity.

Tools should be assessed according to their type. Combined software production and verification tools should be assessed in accordance with the guidelines for software production tools unless partitioning between the two functions can be demonstrated. Only deterministic tools may be assessed, that is, tools which produce the same output for the same input data when operating in the same environment. The assessment process may be applied either to a single tool or to a collection of tools.

5.7 Code analysis

Responsibility: *Software Manager and Software Engineer*

Analyses should be conducted as described in Section 2.5.

Code analysis should be performed prior to software (module) integration. It purpose is to detect and report on faults that may have been introduced during the software programming process. This analysis should confirm that the products of the programming process are accurate, complete and can be verified. Primary concerns include the correctness of the code with respect to the requirements and the software architecture. This analysis is usually confined to the source code.

The analysis should investigate:

a) Compliance with the low-level requirements: The objective is to ensure that the code is accurate and complete with respect to the software low-level requirements.
b) Compliance with the software architecture: The objective is to ensure that the code matches the data flows and control flows defined in the software architecture.
c) Ability to be verified. The objective is to ensure that the code does not contain statements or structures that cannot be verified (for example, code that must be altered in order to test it).
d) Accuracy and consistency: The objective of these analyses should be to consider stack usage, fixed point arithmetic overflow and resolution, resource contention, worst case execution timing, exception handling, identification of variables that are used but not set, identification of variables that are not used, data corruption due to task or interrupt conflicts, etc.

The output from code analysis is passed to the software quality team, verification and validation team.

5.7.1 Software integration analysis

Software integration analysis should be performed prior to verification and software testing. Its purpose is to ensure that the inputs to the integration process are complete and correct. The report or diary of the integration process should include consideration of incorrect hardware addresses, memory overlaps, missing code components, etc. This could be done by a detailed examination of linking and locating data.

The outputs from software integration analysis is passed to the verification and validation team, software quality team, and design review team.

5.8 Logic and algorithm reviews

Responsibility: *Software Manager and Software Engineer*

Before dynamic testing (see Appendix M) begins the code must be reviewed in accordance with the *Software Verification Specification* to ensure that it does conform to the *Software Module Specification*. Indeed it is often very revealing to try to read another person's software. One should be able to relate the code to the design and hence to the specification with little or no difficulty. More often than not this is impossible due to poor documentation, an inadequately defined programming language, poor programming or all three!

There are two different methods for assessing software; walkthroughs (see Appendix N) and code reviews (see Appendix O).

5.9 Software code and supporting documentation

Responsibility: *Software Manager and Software Engineer*

This refers to all code listings and any other design material which provides extra information about the code produced (see Section 2.8.1).

6. Testing

6.1 Testing objectives

Responsibility: *Software Engineer*

6.1.1 Software testing

Testing has two complementary objectives. One is to demonstrate that the software performs to its requirements. The second objective is to demonstrate, with a high degree of confidence, that errors which could lead to unacceptable failure conditions (as identified in the *Hazard Log*) during the service life of the system, have been identified and corrected.

There are two complementary forms of test for a program, namely static analysis and dynamic test. Static analysis (see Appendix J), which does not require the execution of the program, is an analysis of the structural complexity of the code, of the program semantics and possibly a full verification. Dynamic test (see Appendix M) executes the program under typical environmental conditions and analyses the output against expected results.

6.1.2 Testing philosophy

To meet the objectives set out in Section 6.1.1, the testing process should be based principally on the *Software Requirements Specification*. To demonstrate correct functionality, a set of test cases appropriate to the function should be defined. The determination of which test cases to select should be influenced by the second test objective; that is, to establish conditions wherein the presence of a potential error would be revealed. The reason for the emphasis on requirements-based testing is that this strategy has been found to be most effective in revealing errors.

Test cases should be generated to provide coverage of all the *Software Requirements Specification*. Three techniques which relate to different (but overlapping) error sources are:

a) Low-level testing: To verify the low-level requirements.
b) Software integration testing: To verify the interrelationship between requirements and to verify the implementation of the requirements within the software architecture.
c) Hardware/software integration testing: To verify correct operation of the software in the target hardware environment.

Where a test case and its corresponding test procedure can be developed for use during software integration testing or hardware/software integration testing with appropriate requirements or structural coverage, it is not necessary to duplicate the test at a low level. Tests performed at a low level may be less effective than those performed after integration even though they are nominally equivalent. This is due to the reduced functionality available during the low level testing.

Requirements coverage analysis should identify deficiencies in the requirements-based tests. To ensure that the requirements-based tests have adequately exercised the software structure, structural coverage analysis should be completed. Details of these analyses are presented in Section 6.2.3.

6.1.3 Test environment

An ideal test environment includes the fully integrated software loaded into the target hardware, and tested in a high fidelity simulation of the target environment. Certain types of faults can only be revealed in this fully integrated environment, and so a number of tests should be performed at this stage. In many cases, the requirements and structural coverage required can be achieved only with more precise control and monitoring of the test inputs and code execution than is generally possible in a fully integrated environment. Such low-level testing may need to be performed on a small code component which is functionally isolated from the remainder of the software.

In some circumstances it may be possible for testing to be done using a target processor emulator, a simulator or a host computer. The differences between the test environment and the application environment, and the effect of those differences on the ability to detect errors, should be documented in the *Software Verification Specification*. This specification should also document how coverage of those errors is provided by other verification processes. A target emulator or host computer/simulator used for testing may need to be qualified as described in Section 5.6.

6.2 Testing process

Responsibility: *Software Engineer*

6.2.1 Test case selection

To implement the testing philosophy stated in Section 6.1.2, two categories of test cases should be considered, normal range and robustness (abnormal range). The specific test cases should be developed from the requirements and the error sources particular to the software development process.

6.2.1.1 Normal range test cases

The following are representative normal range test cases:

- for functions with execution-time performance requirements (such as filters, integrators and delays), multiple iterations of the code should be performed to check the overall characteristics of the function in context and under different conditions
- the operational software can be viewed as assuming a number of distinct modes, sub-modes or states. Transition conditions from one state to the next

should be defined in the *Software Requirements Specification*. This logic should be tested by inputs which establish all operational states and which exercise all state transitions possible during normal operation

- for requirements expressed by logic equations, the objective is to verify the variable usage and the Boolean operators. One method to achieve this objective is to test all combinations of the variables (multiple-condition coverage). For complex expressions this method is impractical due to the large number of test cases required and a different approach, which ensures adequate coverage, is needed, e.g. the operators could be verified by analysis or review, and to complement this activity, test cases could be established to provide modified condition/decision coverage. Another good solution to this problem of scale is to use "intelligent exhaustion", a technique developed at the Royal Signals and Radar Establishment (RSRE), Malvern. This uses logical reasoning to reduce the test space, where, for example, mutually exclusive events are not tested in their various combinations.

6.2.1.2 Robustness test cases

The test cases chosen should include a demonstration of the ability to deal with abnormal inputs.

The following are examples of typical robustness test cases:

- real and integer variables should be exercised using equivalence class selection of invalid values. The possible failure modes of all incoming data should be considered. Particular attention should be given to complex digital data strings from an external system
- for loops where the loop count is a computed value, test cases should be developed to attempt to create out-of-range loop count values, and thus demonstrate the robustness of the loop related code
- for time-related functions, test cases should be developed to check exceeded frame time protection mechanisms and initialisation during abnormal conditions
- for state transitions, test cases should be developed to provoke transitions that should not occur according to the *Software Requirements Specification* and to demonstrate that such transitions are appropriately handled
- some hardware faults (e.g. EMC interference, open circuit wires) may not necessarily stop the processor. Test cases should be developed to simulate such conditions.

6.2.1.3 Boundary value analysis

Input data can often be divided into a number of equivalence classes, for which all the data within that class should behave in a comparable way. Output can also be divided in a similar manner, and test data should be found to represent all the input equivalence classes and to (try and) force the production of all the output equivalence classes.

Programming or algorithmic mistakes are often made due to a misunderstanding of the desired behaviour at the boundaries of equivalence classes. Boundary value analysis is concerned with the identification of the test data necessary to verify that these boundaries have been implemented correctly.

6.2.1.4 Error guessing

Error guessing is an experience based activity in which, in effect, an engineer inspects the code, and by a mixture of experience and intuition identifies the weaknesses and writes test cases to expose those weaknesses. However, being a somewhat intuitive process, it is very difficult to specify or define it as a procedure.

6.2.2 Operational modes

6.2.2.1 Functional tests

These tests are used to ensure that the specified functions under normal operational conditions, disturbed operational conditions, and extreme environmental conditions are performed fault-free. Functional testing examines the functions in the *System Requirements Specification* and the system design. The former are checked during the system and acceptance tests, and the latter during integration testing. It is anticipated that a function of the *System Requirements Specification* should be implemented by a function in the system design.

Most of the functions of programmable safety-related systems can only be tested if the software is integrated into the system. The amount of testing depends on the integrity level (see [1]). Ideally a complete functional test (**exhaustive** testing) should be made on the integrated system. However, this cannot normally be done in practice. One way around this problem is to use **intelligent exhaustion** where logical reasoning is employed to reduce the test space (see Section 6.2.1.1). In the following the two typical operational modes are briefly described.

a) *Normal operation* — The input signals are varied according to the specification, and the output signals are monitored and compared with the specification.
b) *Disturbed operation* — These tests are used to monitor the behaviour of the system if a failure, or an illegal data combination occurs, or if specified values are exceeded. It shall be ensured that all possible hazardous failures are recognised and appropriate safety reactions are performed.

The following may be simulated. Failure of the:

- technical process
- safety-related system
- self-test routine
- input/output signals (exceeded values, illegal combinations, etc.)
- sensors
- actuators

- operator
- energy source.

c) *Extreme environmental conditions* — The functionality of the system must be tested under extreme environmental conditions such as climate, electromagnetic influences, shock, etc.

6.2.2.2 Simple and extended functional tests

Simple functional tests check whether the specified behaviour of the system is maintained under normal operational conditions with simple simulations of external faults. The input signals are varied under defined conditions, and the output is monitored and compared with the specification. In extended functional tests external faults are simulated and the behaviour under extreme and unusual operational conditions is tested.

6.2.3 Coverage analysis

Ideally one would like to say that all possible inputs have been applied to the system. For even a relatively simple program this is obviously impractical, and thus a decision has to be taken as to when to stop testing.

Unfortunately there is no definitive cut off point to the testing process although a number of commercially available test systems do attempt to provide metrics for "test coverage". These metrics relate to code coverage (statements, branches or paths), system coverage (procedures analysed) and interface coverage (input/output operations). Determining an acceptable coverage for a given risk is difficult. It has been argued [24] that the achievement of adequate levels of testing for safety-related software is beyond current technology. [25] argues that assurance in a system rests upon dynamic testing (see Appendix M), mathematical review of all development stages, and the certification of personnel and the process. They also point out that the distribution of the test data must be representative of the operating environment if the testing is to be statistically valid. Indeed, systems whose "trajectory" of input data is long (i.e. where system behaviour can depend upon events arbitrarily far in the past) cannot be satisfactorily tested [25].

Unless a test strategy is well planned and documented, showing an intelligently selected test data set complete with justifications for the tests, then little confidence can be placed in the test procedures.

6.2.3.1 Test coverage analysis

Test coverage analysis is a two step process, involving requirements coverage analysis and structural coverage analysis. The first step analyses the test cases in relation to the *Software Requirements Specification* to provide confidence that the test cases selected meet the approved criteria. The second step provides confidence that the requirements-based test procedures adequately exercise the code structure.

6.2.3.1.1 Requirements coverage analysis

The requirements coverage analysis should show that:

a) Test cases exist for each requirement.
b) The test cases are adequate to test each requirement, i.e., the test cases meet the approved criteria based on the concepts of normal and robustness testing presented in Section 6.2.1.

This analysis may reveal the need for additional requirements-based test cases.

6.2.3.1.2 Structural coverage analysis

The requirements-based test cases may not fully exercise the code structure, so structural coverage analysis should be performed. The objective of this analysis is to identify the code structure which was not adequately exercised by the requirements-based test procedures. The analysis should demonstrate coverage to the modified condition/decision level. Data coupling and control coupling between the different code components should also be exercised.

The structural coverage analysis may be performed at the source code level, except where the compiler generates object code which is not directly traceable to source code statements (for example, a compiler-generated array-bounds check in code). In such cases, additional verification to establish the correctness of such generated code sequences should be performed at the object code level.

Structural coverage analysis may reveal structure which was not exercised during testing. This may be the result of:

a) Shortcomings in requirements-based test cases or procedures: The test cases should be supplemented or the test procedures changed to provide the missing coverage. The methodology used to perform the requirements-based coverage analysis may need to be reviewed.
b) Inadequacies in the *Software Requirements Specification*: The *Software Requirements Specification* should be modified accordingly and additional test cases developed and test procedures performed.
c) Unreachable code: The code should be removed and an analysis performed to assess the need for re-verification. Where redundant code exists for a valid reason this must be justified and documented.
d) Deactivated code: For deactivated code which is not intended to be executed in any operational configuration, a combination of analysis and testing should show that the means by which such code could be inadvertently executed are prevented, isolated, or eliminated. For deactivated code which is only executed in certain configurations of the target system, the operational configuration needed for normal execution of this code should be established, and additional test cases and test procedures developed to meet the required coverage objectives.
e) Limitations of the test environment: For code which cannot be executed under the test environment, a combination of analysis and testing should be used to demonstrate its

correctness.

6.2.3.2 Exhaustive input testing

There are situations where the software portion of a system is simple and isolated such that the entire input and output space can be bounded. In such cases, it is possible to demonstrate that exhaustive testing of this input space can be substituted for software verification.

In such cases the following should be demonstrated:

a) An analysis which provides a complete definition of the input/output space of the software.
b) An analysis which demonstrates the isolation of the inputs to the software.
c) Rationale for the exhaustive input test suite.
d) The test procedures and results.

6.2.4 Error seeding

These techniques are used to justify the effectiveness of the testing process. Errors are introduced at strategic locations in the software by the insertion of faults. The standard tests are then performed on the software with the intention that all, or an acceptable number, of these known faults are identified. Under the assumption that seeded faults are discovered with the same consistency as unseeded faults, this process gives an indication of the level of coverage of the testing process adopted.

Error seeding is also useful as a way of testing the effectiveness of error detection and recovery routines.

6.2.5 Statistical methods

Traditional reliability analysis is performed to assess the effects of random hardware failures. However, since software failures are entirely attributable to development faults, software reliability evaluation concentrates on the development process. It should be noted that since the final result of such an analysis is to measure the reliability of the entire system, the metrics for software reliability must be comparable to those of the traditional hardware model.

Reliability growth models rely upon predicting future behaviour by analysing past behaviour. Thus such techniques are only applicable towards the end of the development. It is sometimes felt that such techniques are not applicable for the development of safety-related software. The reason for this is that after a successful development of safety-related software, so few faults should exist as to make the probabilistic model virtually meaningless.

6.3 Integration

Responsibility: *Project Engineer and Software Engineer*

Integration covers:

- combination of individual software modules to create the overall code
- merging of the software system onto the target hardware
- overall system integration.

A *Software/Hardware Integration Plan* shall be produced which clearly distinguishes those activities. The plan shall be developed throughout specification, design and development phases and shall document the following:

- the split of the system into integration levels
- types of test to be performed
- test cases and test data
- test environments (tools, support software, etc.)
- criteria for judging successful completion of the tests
- requirements for independent assessment.

The *Software Integration Test Results* should state the results of the tests and whether the objectives and criteria of the *Software Integration Plan* section of the *Software Design Plan* have been met. The reasons for any failure should be fully recorded. Particular attention should be paid to the functionality of the completed system. This material should be produced in a form suitable for audit by the assessor.

6.3.1 Module integration testing

This technique of testing may be performed by increasing the scope of requirements through successive integration of code components with a corresponding expansion of the scope of the test conditions.

Typical errors which may be revealed by this testing technique include:

- incorrect initialisation of variables
- parameter passing errors
- data corruption (especially global data)
- inadequate end-to-end numerical resolution
- incorrect sequencing of events and operations.

6.3.2 Hardware/software integration testing

This technique of testing should concentrate on the high-level functionality of the software, and on error sources associated with the software operating within the target environment.

Typical errors which may be revealed by this test technique include:

- incorrect interrupt level handling
- failure to meet execution time requirements
- incorrect software response to the hardware transients or hardware failures (for

example, start-up sequencing, transient input loads and input power transients)
- bus or other resource contention problems (for example, memory mapping)
- inability of built-in test to detect failures and faults
- errors in hardware/software interfaces
- incorrect behaviour of feedback loops
- incorrect control of memory management hardware, or other hardware devices under software control
- stack overflow
- incorrect operation of any mechanism(s) used to confirm the correctness and compatibility of loadable software
- violations of software partitioning.

6.3.3 Functional validation

Functional validation is carried out for two purposes:

- to ensure that the specification of functional requirements is complied with
- to ensure that requirements imposed by, for example, legislation are met.

The latter usually takes the form of a series of tests specified in the relevant legislation, the results of which are formalised and used in a Type Approval or other certification submission. Such tests are not designed to ensure that software is fault free, but rather to demonstrate that the system performs under specified conditions as required by the certifying authority.

The former case is somewhat different. The purpose is to ensure that all the provisions of the *Software Functional Requirements* are complied with, and, for example, that under all normal operating conditions the system performs the desired task. This may involve both rig and vehicle testing, which may be carried out over a significant period of time, with the system subjected to a set of stimuli either under automatic control, or test driver control, which will thoroughly demonstrate that the system does actually do what is expected of it.

Abnormal conditions may also be tested, and will be expected to produce the desired response to the same degree of confidence as normal operating conditions. However, abnormality will normally be confined to what is possible by the disablement of sensors, environmental tests including EMC, and the limitations of test rig or vehicle driveability technology.

6.4 Test documentation

Responsibility: *Project Engineer and Software Engineer*

6.4.1 Software acceptance plan

The *Software Acceptance Plan* describes the process to be used for accepting the software. This is particularly important if the development has been sub-contracted (see ST7 Sub-contract).

6.4.2 Software acceptance test specification

The software part of the sub-system will normally be developed by a specialist team, either in-house or sub-contracted to another company. A plan must therefore be made as to when the software will be considered to be complete in all respects; including the programming, testing and documentation (see Section 6.4.1). The *Software Acceptance Test Specification* details the tests that will be performed as part of the acceptance process and which both the customer and the developer believe adequately reflect the requirements.

The result of this phase should be the *Software Acceptance Test Specification*, which must be agreed by both parties.

6.4.3 Software module and integration test specifications

As testing is a major part of the verification and validation process it must be planned accordingly. Testing should be performed at two levels. Initially each module should be tested separately, and then, once the modules have been integrated to create the system, tests should be made on the entire system. The purpose of module testing is to show that each module both performs its intended function, and does not perform unintended functions. The purpose of testing the entire software sub-system is to ensure that all modules interact correctly to perform their intended function, and do not perform unintended functions.

The degree of independence between the software designers/implementors and the test planners will depend on the required integrity level (see Section 2.7).

Separate testing should also be performed as part of the development process to determine whether the specified safety requirements are fulfilled. A distinction must be made between the *Software Functional Safety Requirements*, which specify those functions that must be carried out by the safety-related control system should a hazardous error occur in the technical process, or in the safety-related system itself, and the *Software Safety Integrity Requirements*, in which the measures to avoid and to control hardware failures are described. The basis for testing is the *Test Plan* section of the *Software Validation Plan*, and all activities and results must be documented in such a manner, that they can be used for further evaluation processes.

For each phase of the life-cycle of system development, as well as for the validation of the final system, a *Test Plan* section of the *Software Validation Plan* must be created. The result should be the *Software Integration Test Specification*, together with a number of entries in the *Software Module Test Specification*.

7. Assessment

In this document the term "assessment" is used to cover the range of activities from an audit up to a full assessment.

7.1 Safety case

Responsibility: *Software Manager*

The safety case is the documentation of the reasons why the system is believed to be safe to be deployed and it reflects the design and assessment work carried out in the development process. In many situations, the safety case (or a summary thereof) will be the major deliverable to the certification process or to an independent assessment.

The safety case [26] comprises four elements each of which demonstrates a different aspect of competent safety management. The elements are:

i) A description of the system and the activities to be carried out. The object of this element is to demonstrate that the system has been properly designed for the operations that are to be carried out, and inherent in this is a demonstration that the system is built to established codes and standards.
ii) An analysis of potential major hazards. A demonstration is required that *all* hazards, as recorded in the *Hazard Log*, have been considered and that a quantitative evaluation of the risks has been carried out (see also [1]).
iii) A demonstration of the management systems in place that recognises the hazards and reduces the risk to one that is acceptable or As Low As Reasonably Possible (ALARP).
iv) A justification for continued operation. This must demonstrate that where risks from a hazard are unacceptable, measures are in hand to reduce them ALARP as soon as reasonably practicable. This will necessitate arguments as to why the anticipated operation of the system, given the anticipated inputs and failures, will not lead to the identified hazards. In practice, the arguments will reflect design information, reliability calculations, safety analyses, and perhaps proofs of programs properties. The *Hazard Log* acts as the index as it should be possible to find the safeguards against each particular hazard, and the reasons for believing them to be effective, by tracing from the hazards.

7.2 Leading to acceptance

Responsibility: *Software Engineer*

The TickIT guide [2] suggests three forms of acceptance; first party acceptance by the supplier, second party acceptance by the purchaser and third party acceptance by an independent auditor. The latter would be required in order to be granted certification.

7.2.1 First party (supplier) acceptance

There are two scenarios for first party acceptance. The first is when a vehicle manufacturer produces its own software which is assessed internally.

The second is when the software is produced by a sub-contractor (see [5]).

In this case the supplier will have a good knowledge of the system and so it may be useful if they are involved in writing parts of the *Software Acceptance Plan*, running the tests, and evaluating the results. This form of acceptance relies upon the purchaser trusting the supplier and the supplier having a reliable internal quality management system. This form of first party acceptance may occur in several stages both before and after delivery.

The purchaser may still wish to check through the *Test Plan* section of the *Software Validation Plan* and supervise the execution of the tests and their evaluation.

7.2.2 Second party (purchaser) acceptance

When the supplier has completed the validated product, the purchaser or the eventual users of the system should judge whether or not the product is acceptable according to previously agreed criteria and in a manner specified in the contract. Acceptance testing usually occurs in two phases:

- at the supplier's site before delivery,
- at the purchaser's site after installation.

Although for formal contractual reasons some purchaser acceptance may be needed, it is difficult to justify this in technical terms if the purchaser has already indicated acceptance of the natural language specification (and perhaps the executable specification (see Section 3) as well). Non-acceptance implies that either the natural language specification is ambiguous (all too likely) or that it was incorrectly interpreted when producing the formal specification. A purchaser review of the executable specification is obviously advantageous in reducing the risk of non-acceptance of the final software.

The method of handling problems detected during the acceptance procedure and their disposition should be agreed between the purchaser and supplier and should be documented. It can be difficult to persuade the supplier to rectify any faults discovered during usage if this is after the purchaser has formally accepted the product.

Since second party acceptance involves the purchaser with the system this process will also form the initial phase of user training. However, this lack of knowledge of the system may cause difficulties during the acceptance process.

7.2.3 Third party (auditor) acceptance and certification

An independent organisation may be employed to provided an unbiased form of acceptance.

When this organisation is a representative of the relevant regulatory body then this form of acceptance can form the basis of certification. Certification is a legal recognition, by a certification authority, that a system complies with regulatory requirements. Normally this will include the formal process of recording that the safety requirements have been met, that in the opinion of the independent regulatory body the safety-related software is of adequate safety integrity [1], and that all the work associated with the creation and provision of the safety-related software has been diligently carried out in accordance with the requirements of the MISRA Guidelines.

7.2.4 Acceptance tests

The Software Tools for Application to large Real-Time Systems (STARTS) Purchasers' Handbook [27] suggest the following as typical acceptance tests:

- correct function tests
- incorrect, abnormal or error path function tests
- performance testing:
  - timing tests
  - capacity and volume tests
  - stress tests
  - endurance tests
  - graceful degradation
- security tests
- recovery tests
- other quality attribute tests:
  - usability
  - adaptability
  - availability
  - reliability
  - maintainability
- configuration testing
- documentation testing
- installation testing
- state tests
- pilot and parallel running
- alpha and beta testing
- human factors.

7.2.5 Product acceptance

The product acceptance phase of the software system development is normally the last development life-cycle activity to be performed. It normally marks the end of the development and the beginning of actual use of the system. In order to be successful and run smoothly, this phase must be planned well in advance.

The acceptance criteria for safety-related software should distinguish between errors in formal

argument, and discrepancies found during dynamic testing. The criteria should include the maximum number of errors found by the verification and validation team in formal arguments or during static path analysis beyond which the item will be redeveloped from scratch. Errors found by dynamic testing in properties verified by formal arguments are particularly serious, and would generally lead to non-acceptance of the application. There are issues which could make a product which passes the comparison testing (see Section 6) unacceptable such as performance. These issues need to be identified in advance and criteria produced to reduce the risk. This is not easy, since, for example, predicting the performance of anything other than trivial systems is notoriously difficult.

Many safety-related systems may have to be formally accepted by a regulatory authority which may have statutory responsibilities for ensuring that the system meets predefined levels of safety. Within a statutory framework, there is usually the facility to allow systems which provide safety to do this in many different ways. The regulatory authority usually requires that the supplier produces documentary evidence of the design and the processes used in formulating the design.

To ensure that procedures are observed, many of the activities may be controlled by a safety authority (board) which monitors the development process and considers requests for deviations. Any deviation granted will be recorded for further review by the regulators.
Even if a particular system does not need to be formally certified by a regulatory authority, the existence of product liability legislation suggests that a company should proceed along similar lines using its own "assessor" (see [1]) as the certifying authority.

Third party assessment may be difficult when there are intellectual property rights restrictions.

To ensure that the third party certificate is acceptable to the purchaser, and any other potential purchasers, the issuing body must have some form of recognised accreditation.

The STARTS Purchasers' Handbook [27] has a substantial chapter devoted to this subject.

7.3 Risks and limitation of assessment

Responsibility: *Project Manager and Software Engineer*

Whilst the use of a rigorous argument will effectively identify many design errors without the time consuming process of performing a formal proof, it must be stressed that a rigorous argument is no substitute for a formal proof. Ultimately the acceptance of a safety-related system will rest on its safety case whose degree of rigour will depend on the integrity level required (see [1]).

New techniques will have to be judged individually against the confidence that can be placed on current techniques. Thus whilst object-oriented design, or the use of artificial intelligence, currently have some drawbacks with respect to their use in safety-related applications, some techniques having more drawbacks than others, the situation can be expected to change in the

light of current and future research.

7.4 Assessment analysis

Responsibility: *Software Manager and Software Engineer*

Analyses should be conducted as described in Section 2.5.

7.4.1 Verification and test results analysis

Verification and test results analysis is part of the validation process.

Its purpose is to ensure that the output of the software programming process is compliant with the *Software Requirements Specification,* and that testing is defined and carried out to

a) Identify programming errors and inconsistencies.
b) Carry out test coverage analysis.
c) Identify and document any software constructs which may cause deviations from the *Software Requirements Specification*.

The report should detail the test results and the actions taken, or to be taken.

The outputs from verification and test results analysis is passed to the design review team, software quality team.

7.4.2 Review of the products of the verification and validation process

The purpose of the verification and validation process is to establish a level of confidence in the software: that it is essentially "bug"-free; that it does what its specification lays down; that its specification does what its customer intended, no more, no less.

A number of pre-defined tests (see Section 6) will have been carried out on the software, normally at module or routine level, rather than as a whole, using software test tools. From these tools a level of confidence may be established; this is usually expressed in terms such as test coverage. There are many ways to express test coverage (e.g. using the number of branching paths to record the percentage of the structure that is exercised, or by seeding paths with counters to record if a path has been executed by a test, or by employing a complexity measure).

The output from the verification and validation process will be a formal report, which will identify what level of confidence has been achieved (see [1]), what tests have been carried out, how many errors notified, any deviations from specifications, and any deviations from the customer’s expectations. The review of the report should consider whether the level of confidence achieved is acceptable, whether and how it may be improved, and the impact of any deviations from specifications or customer expectations. The output from this review

should be either a requirement for identified shortcomings to be resolved by actions agreed by the review team, or else result in sign-off of a software acceptance document.

7.5 Verification and validation documentation

Responsibility: *Software Manager and Software Engineer*

7.5.1 Software requirements verification results

The *Software Requirements Verification Results* addresses the adequacy of the *Software Requirements Specification* (see Section 3.4) in meeting the requirements assigned to the sub-system by the *System Requirements Specification*.

7.5.2 Software design verification results

The *Software Design Verification Results* section of the *Software Verification Results* (see Section 2.4.5.2) addresses the adequacy of the *Software Detailed Design Specification* (see Section 4.4.3) in meeting the *Software Requirements Specification* (see Section 3.4).

7.5.3 Software module design verification results

The *Software Module Design Verification Results* addresses the adequacy of the *Software Module Specification* (see Section 4.4.4) in meeting the *Software Detailed Design Specification* (see Section 4.4.3). It should also confirm that the *Software Module Test Specification* and the *Software Integration Test Specification* (see Section 6.4.3) are suitable to test that the *Software Requirements Specification* has been achieved.

7.5.4 Code review results

At the end of code review, the reviewer's comments should be recorded in the *Code Review Results*, and the resulting actions followed through and documented under full change control (see Section 2.10).

7.5.5 Software module test results

Once the code for a module has been written it can be tested in accordance with its *Software Module Test Specification* (see Section 6.4.3). This will include both static and dynamic testing.

The result should be the *Software Module Test Results*, which should contain the following information:

- the hardware and software configurations used and their version numbers
- input test data
- output test data

- additional data regarding timing, sequence of events, etc.
- the degree of conformance with the acceptance criteria given in the *Software Module Test Specification* (see Section 6.4.3).

7.5.6 Software module error incidence results

A *Software Module Error Incidence Results* document may be maintained; these describe the character of each error, and the action taken by the design team.

7.5.7 Software integration test results

Once each module has been successfully tested on its own, they can be put together to form the Integrated Code. This should then be tested in accordance with the *Software Integration Test Specification* (see Section 6.4.3). This should contain the same type of information as for the corresponding *Software Module Test Results* (see Section 7.5.5).

The result should be the *Software Integration Test Results*.

7.5.8 Software integration error incidence results

Software Integration Error Incidence Results should contain the same type of information as for the corresponding *Software Module Error Incidence Results* (see Section 7.5.6).

7.5.9 Software validation results

The *Software Validation Results* should identify the hardware and software used, the equipment used and its calibration, the simulation models used, and any discrepancies and corrective actions taken. The report should summarise the results of the validation and should assess the system's compliance with all requirements.

Any software tools and simulations used in the validation process should be identified as an item in the *Software Validation Results*.

7.5.10 Software acceptance test results

Once the software team believe that they have completed the software in all respects, the client should be informed so that the acceptance tests can be performed in accordance with the *Software Acceptance Test Specification* (see Section 6.4.2).

The result of this exercise should be the *Software Acceptance Test Results*, which should be agreed by both parties.

8. Maintenance

8.1 Software maintenance

Responsibility: *Software Engineer*

Software maintenance has been defined [28] as:

> The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.

Whilst hardware maintenance is usually required because of the progressive degradation or wearing out of physical materials, software maintenance is performed for one of four reasons (see [29]); perfective (50%), adaptive (25%), corrective (21%) or preventive (4%):

- perfective maintenance is performed to bring the software into line with a change in the *User Requirements Specification*
- adaptive maintenance is performed to bring the software into line with a changing performance requirements or changing operating environment
- corrective maintenance is performed to eradicate faults in the software
- preventive maintenance is occasionally used to describe work done to prevent malfunctions or to improve maintainability.

The maintenance phase of a project is normally considered to be a separate phase of the system life-cycle and is often funded in a completely different way from the development phases. It is characterised as being made up of a number of small development steps, each one adding to, or modifying, small features of the system. Thus a similar set of methods and tools to those used during development will be required during this phase.

Most software evolves, and often the more successful software is, the more that users will demand modifications and changes to it. By its very nature safety-related software will interact heavily with its environment which will inevitable be constantly changing. The heavy costs of producing safety-related software produces a desire for the software to be long lived and hence such software should be built with maintainability in mind, and quality assurance must continue to sustain or enhance its quality throughout the whole lifetime of the product.

For a design to be maintainable, it must be constructed in a modular and structured manner so that it is readily adaptable (one such way is to use an object-oriented approach (see Appendix K)). Thus each component should be as self-contained as possible, and have a minimal reliance on external components (e.g. system functions or error handling functions). If during maintenance, there are adaptations which involve changes to a component's interfaces then the relevant external functions must also be considered. For further adaptability the design should be well documented, the documentation should be readily understandable and consistent with the implementation, and the implementation should be

expressed in a readable way. It is important to stress that all software items, including specifications, test suites, and documentation must also evolve in step.

Maintenance can be carried out by:

- the developer
- the purchaser or related organisation
- an organisation different from either of the above.

The second and third options imply that a facility, similar to the one used in the development phase, will need to be set up for the maintenance phase. It will need to contain all the relevant tools used in development; these may possibly include some built by the developer.

One objective of software maintenance is to make the corrections, enhancements or adaptations to the software, whilst ensuring that the required integrity level is maintained. A controlled cycle of procedures for the maintenance of software should be established and recorded in suitable documentation. These procedures should include:

i) Control of error reporting, error logs, maintenance logs, change authorization and software/system configuration.
ii) Testing, verification, validation and assessment.

A *Maintenance Record* should be established for each maintenance activity. This record should include:

i) The modification or change request.
ii) An analysis of the impact of the maintenance activity on the overall system, including hardware, software, human interaction and the environment and possible interactions.
iii) The detailed specification of the modification or change.
iv) Re-validation and regression testing (see Section 8.7) of the modification or change to the extent required by the software integrity level.

A *Maintenance Log* should be established and maintained. It should include:

i) All change requests and approvals.
ii) Progress of changes.
iii) Change consequence information.
iv) Test cases for components including re-validation and regression testing data.
v) Software configuration history.
vi) Deviation from normal operations and conditions.

Maintenance should be performed with the same level of expertise, automated tools, planning and management as the initial development of the system.

Maintainability should be designed into the software system, in particular, by following the design and development requirements of Section 3. Maintenance control procedures for the software should be considered during the design process to avoid any negative impact on

integrity. The control procedures may be organised into a hierarchy of one or more control levels, internally or externally. The procedures should be strictly defined, and audited at regular and frequent intervals.

Unless justified in the *Quality Assurance Plan* the maintenance control procedures in Figure 8.1 should be used, and associated documentation should be produced.

One of the specific problems associated with maintenance relates to the sporadic nature of the maintenance activity. It may not be continuous and there will be periods when the system is not touched. This means that, when the software engineers do start to work, they will need to revise their knowledge of the system and development procedures. In these circumstances, it is especially important that the procedures are automated to ensure that a consistent set of procedures is adhered to; otherwise, the system's life may be shortened because it becomes impossible to maintain.

8.2 Problem reporting reviews

Responsibility: *Project Engineer and Software Engineer*

The most rigorous approach to software creation offers no more than a level of confidence that it is error-free: problems may be reported when the software is in service. In highly critical systems, it is expected that there will be a problem reporting and action procedure, which may require a periodic review (see Section 2.5) of the problems reported, and the action taken to correct them, as part of the *Maintenance Procedures* for the system.

The resolution of any problem may involve significant re-validation in a critical system, since **any** problem, however minor, may carry a serious implication regarding re-working of the software, with a corresponding implication for the integrity of the software. If a problem is reported which indicates a need for a significant re-working of the software, it is desirable to convene an ad hoc review team meeting to examine the issue and all its implications. It may also be desirable to convene a review in the case that more than a specified number of problems, or serious problems, are reported, for an item of software.

8.3 The user interface

Responsibility: *Software Engineer*

Often, the user interface, although meeting specified requirements, is the subject of user complaints. It does what is expected of it, but what is expected of it proves less than desirable in service.

It should be borne in mind that modifications to the user interface may have far-reaching implications for the system's software integrity. It is advisable for modifications to the user interface to be referred to a review team.

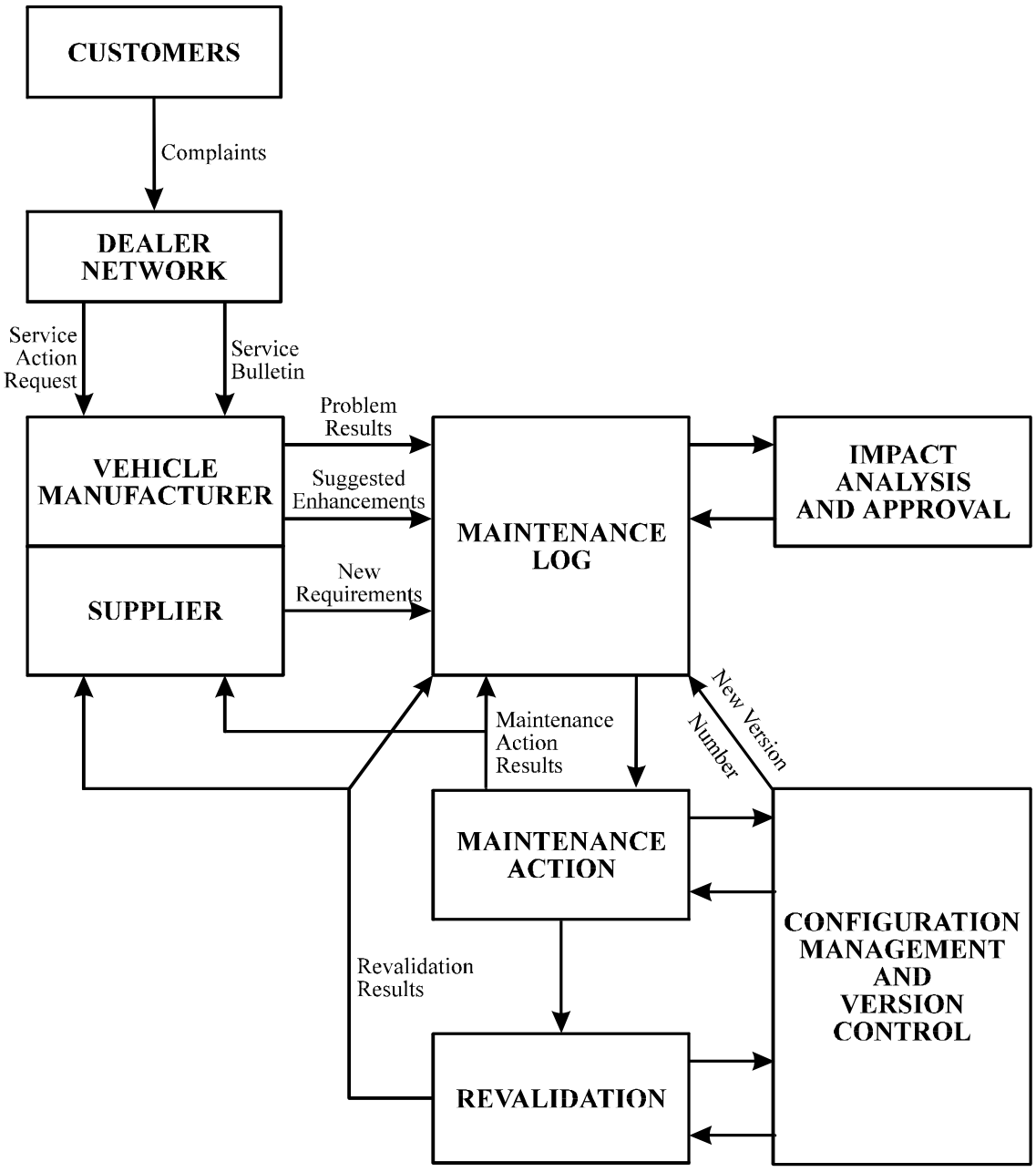


Figure 8.1 - Maintenance Cycle

8.4 Maintenance in safety-related software

Responsibility: *Software Engineer*

It is important that the integrity of a safety-related system is not affected by poor *Maintenance Procedures*. Each change to a safety-related system must go through the same rigorous life-cycle of development as the original system, that is, a change to the requirements must initiate a full analysis of the whole *System Requirements Specification*, a change to the design will require a full verification and validation analysis, etc.
Experience has shown that even what may be considered to be minor changes to the "code" often have quite dramatic and unexpected effects on the system's operation. Any change to the system, whether it be hardware or software, should be authorised and the complete test procedures repeated and checked. This is contrary to traditional engineering practices where a developer works within a domain wherein changes exhibit a continuity that enables their effects to be predictable.

There will often be economic or timescale pressures to avoid undergoing a complete life-cycle for each minor change but the temptation to "do a quick fix" must be resisted. If a "quick fix" approach is adopted, all the effort put in to the original system development will be lost.

The replacement of a safety-related system by a new version should require the same levels of authorization and assessment, and follow the same procedures as applied to the original system.

8.5 Configuration management and quality assurance

Responsibility: *QA Manager*

To enable modifications to be made to a software component we require the whole system to be covered by a configuration management system and metrics to provide visibility. This should include an ability to trace a component throughout a hierarchically described design; and to be able to relate the design to its specification. The latter may be ameliorated by using the same decomposition technique for both the design and the specification (e.g. object-orientation (see Appendix K)). The same level of quality assurance must be sustained throughout the maintenance process. Document management should be used to ensure that any changes made to the design propagates throughout all relevant supporting documentation to avoid inconsistencies occurring. Inconsistent documentation makes maintenance much more difficult. Thus a clear management policy is required to ensure that any modifications made to the system do not affect its future maintainability.

8.6 Software re-use

Responsibility: *Project Engineer and Software Engineer*

The notion of re-using software is one that developed at a very early stage in the history of

computing, and was a strong motivation for the development of the use of subroutines.

Given some statement of requirements, or some detailed specification, or even a high-level design, this will need to be matched with the re-usable software held in a library. [7] suggests two main types of solution:

1) Finding a single component which will fully, or almost fully, satisfy the requirements.
2) Finding a collection of components, suitably interconnected, which can satisfy the requirements.

The first case is already a common problem in package selection, although this important part of commercial software practice is almost entirely ignored by system development methodologies. The requirements or specification must be matched against the specifications in the library. (This process often occurs during sub-contracted work (see [5].) The task of matching precise (formal) specifications exactly is in general undecidable, thus descriptions are simplified for search purposes, and partial or approximate matches are accepted along with the further process required to vet the retrieved components to find the most suitable.

In the second case the requirements must be decomposed. One can then either look for some sub-match of library components to the requirements, or divide the requirements into separate parts which can then be searched for in the library.

8.6.1 Software module re-use and software libraries

Many automotive products involve the re-use of software modules, or of software components previously designed and stored in libraries. This indicates four possibilities:

1) software to be re-used may not have been subject to quality procedures.
2) some software in a library may have been subject to quality procedures, some not.
3) software to be re-used may have been subject to earlier, less rigorous, quality procedures.
4) a module which is required for re-use may have been subject to up to date quality procedures, but previously used in a lower criticality application.

Where it is desired to re-use a software component, the issue of quality may not be clear cut; there are many items of software in use that have not been subject to up to date quality procedures, but which have been demonstrably fault free for a significant period of years. Any saving achieved by re-use could be negated by the need to carry out a complete verification and validation exercise for the item to be re-used.

Software to be re-used should be the subject of review. The following questions should be asked:

- what is its service history?
- have problems been reported, and of what seriousness?
- when were problems last reported?
- how was it validated?

- how critical is it to the product in which it is to be used?
- how critical was its previous use?

The review may consider that it is acceptable to re-use the software being reviewed without further action; if so, this decision must be recorded, along with the reasons for the decision.

If the review concludes that the software under consideration needs no additional effort to make its re-use acceptable, this must be justified. The review must examine carefully whether additional effort can achieve the required level of confidence in the item of software being considered, or whether a new piece of software should be created, using the current quality procedures. This may not be a simple decision, as there will inevitably be commercial pressures on the reviewing team to re-use existing software without additional effort being expended; this pressure may be very strong and any decision to re-validate or re-write the software may need to be supported by detailed analysis.

8.6.2 Re-use and retrospective techniques

In principle there is little difference between retrospectively validating a piece of software and validating software for re-use. Probably the most important difference is in the way in which the *Software Requirements Specification* has been written and proved. Unless a formal specification language (see Appendix E) has been used, validation of the *Software Requirements Specification* may only be done by inspection, with the use, for example, of cause-effect graphing (see [6]) or else the *Software Requirements Specification* must be fed into a specification analysis tool, manually, and a formal language specification developed. The latter process may contain its own errors, and considerable care must be taken to ensure that the specification is correctly entered into the analysis tool. This is, however, a costly process, and the former method is probably more likely to be adopted. If carried out with care, creating a cause-effect graph from the specification should help to identify any inherent short comings in its structure.

It is perhaps more useful to use code testing tools to look for programming errors by testing as many paths through the code as possible. It is also a valuable exercise to manually inspect the code to assess whether it has adequate modularity, good structure, acceptable use of e.g. interrupts, partitioning of data, etc.

Above all, it is pertinent to use common sense. A piece of software may not have been validated to the latest standards, but if it has proved reliable in use for several years, is it wise to modify it to meet current practice? The very process of modification may introduce errors which did not previously exist. However, consumer protection law must also be considered (see [5]), since not upgrading software to the "state of the art", and not using "state of the art" tools and proving procedures could be considered negligence in a liability suit.

8.6.3 Assessing "off the shelf" components

Assessment of "off the shelf" components or products falls into two categories:

- using pre-written software

- procuring a complete (sub)system which contains embedded software.

The general procedures are broadly the same in both cases.

8.6.4 Benefits of software re-use

An obvious advantage of software re-use is that costs should be reduced as the number of components that must be specified, designed and implemented in a software system is reduced. Re-use, of course, does not just encompass the re-use of code, it is also possible to re-use specifications and designs. It has been suggested that code re-use is never likely to be cost effective. This is an arguable point, but what is clear is that the re-use of logical software components, in any form, has obvious benefits.

Quantifying (or even guaranteeing) the cost reduction of re-use is difficult because the component might require modification which could cost as must as component development. However, cost reduction is not the only advantage of re-use and systematic software re-use offers a number of advantages:

- *System reliability is increased* — testing (see Section 6) can never be fully comprehensive but along with the re-used component comes its past working history.
- *Overall risk is reduced* — uncertain production costs can be replaced by the more reliable costs involved in re-using an existing component.
- *Effective use can be made of specialists* — specialists can develop re-usable components which encapsulate their knowledge rather than replicating their work with individual projects.
- *Organisational standards can be embodied in re-usable components* — re-use of standard components will enable a consistent "look and feel" for user interfaces to spread.
- *Software development time can be reduced* — re-using components speeds up system production because both development and validation time should be reduced.

8.6.5 Risks and limitations of re-use

Often, automotive systems are based on libraries of hardware and software "building blocks". The vehicle manufacturer will specify the performance required from a particular system; the supplier will then either identify an existing "black box" which may be modified to meet the requirement, or else draw on the library to assemble a suitable "black box" product. Either case will inevitably involve the re-use of software, albeit with additions or modifications.

The ideal case would be that each new "black box" is fully validated, irrespective of the state of validation of individual modules, however, this may not be a viable option, and some reliance will very likely be needed to be placed on adequate procedures to ensure an acceptable level of confidence in the re-use of library routines.

Integration testing is to some extent facilitated by the use of library routines, because the

standard library modules will come together with their own test results, and must be regarded as an important issue in the re-use of software. It should also be remembered that software which has performed faultlessly for a number of years in the field will offer a greater level of confidence in the final product than a wholly new program, however validated.

While standard subroutine libraries have been very successful in a number of application domains in promoting re-use, systematic software re-use as a general software development technique is not commonly practised. There are a number of technical and managerial reasons for this:

- assessing the re-usability of a component in different application domains is difficult since the list of critical component attributes which make a component re-usable are unknown eg. attributes such as environmental independence are essential
- developing a generalised component is more expensive than developing a component for a specific purpose. Thus, developing re-usable components requires an organisational policy decision to increase short-term costs for possible, unquantifiable, long-term gain
- some software engineers often prefer to write components afresh as they believe that they can improve on the re-usable component. This is often true, but at the expense of greater risk and possible higher costs
- there is currently no accepted means of classifying, cataloguing and retrieving software components. The critical attributes of a retrieval system is that component retrieval costs should be less than component development costs.

8.7 Regression testing

Responsibility: *Software Engineer*

Regression testing is applied during system maintenance following any modification to the code. The tests are designed to assess all functions that are effected by the altered code and should be based on the details outlined in the *Test Plan* section of the *Software Validation Plan.*

8.8 Maintenance of libraries

Responsibility: *Software Engineer*

Having identified software which is potentially re-usable and described it in such a way that anyone wishing to re-use it would be able to do so, the problem arises as to how to organise the total collection of all such software and related descriptions. Such a library can be structured by classifying the re-usable software in various ways. A good system of classification not only provides the basis for cataloguing the software, but it also provides a means for finding a particular piece of software held in the library. Large collections of software present similar problems of classification to those well known in library science.

8.9 Maintenance documentation

Responsibility: *Software Manager and Software Engineer*

There are several aspects relating to documentation in the maintenance phase of software. Generally, much of this is linked in with configuration management and documentation control. Specifically:

- authorized access to documents
- the ability to locate the relevant documents quickly
- proper management of modification.

8.9.1 Document retention

Procedures should lay down the period for which documents are to be retained and the regulation of their control during that time. It is important that documentation is available as appropriate for re-use and modifications (see Sections 8.9.2 and 8.9.3), with the proviso that security (see Section 2.10.1) is not compromised. Documents should also be retained with liability issues in mind.

Once the retention period for a document has expired, procedures for disposal may need to be followed to ensure it is irrevocably destroyed, particularly in the case of sensitive material.

8.9.2 Re-use

Parts of software may be re-used at a later stage. It is important that the associated documentation is also carried over with the software. The standard of documentation should be such that the functions and interactions of modules are easily understood in the future by people who may not be the original authors of the specifications or the code. If the software is destined for re-use then the retention period (see Section 8.9.1) for the relevant documentation must reflect the need to keep the documented record of the software throughout its usable life-span.

A suitable format for the documentation for the product is as a Technical Construction File described in the United Kingdom (UK) ElectroMagnetic Compatibility (EMC) regulations (SI/2932 92). As the documentation contained in this file will cover both the hardware and the software, it is necessary to ensure that adequate software documentation is provided. Alternatively a separate software file may be produced if appropriate, although some aspects may be inseparable from hardware (e.g. *System Functional Requirements*).

The software documentation should include:

- system FMEA or hazard analysis
- *Software Functional Requirements*
- detailed specification
- design information showing module verification together with overall

validation against the *Software Functional Requirements*
- software test records showing testing against the design.

Source code or compiler listings need not be included provided the combination of the hazard analysis and the *Verification and Validation Results* are considered adequate by the assessor; however they may be necessary if independent assessment/verification is being considered.

8.9.3 Modifications

It may be necessary to revise software during its lifetime. Much of what was said concerning the control of documents during the development phase is also applicable here. Control of documentation will be linked to configuration management, so that only approved changes to the software are made and are fully documented. Changes to software are likely to require revalidation and this should be documented as for the original validation.

8.9.4 Maintenance procedures

This covers all aspects of maintenance that are identified as the project is developed.

Other related design material:
- the *Maintenance Log*
- the *Maintenance Record.*

9. References

[1] MISRA Report 2, *Integrity*, February 1995.

[2] DTI, *A Guide to Software Quality Management System Construction and Certification using ISO 9001/EN 29001/BS 5750 (TickIT)*, Department of Trade and Industry, 1992.

[3] IEE, *Guidelines for the Documentation of Computer Software for Real-Time Iterative Systems*, Institution of Electrical Engineers, 1990.

[4] IEEE, *Software Engineering Standards Collection*, 1991.

[5] MISRA Report 7, *Subcontracting of Automotive Software*, February 1995.

[6] Myers G L, *The Art of Software Testing*, Wiley Interscience series, 1979.

[7] McDermid J (ed.), *Software Engineer's Reference Book*, Butterworth Heinemann, 1991.

[8] Sadri F and Kowalski R A, "A theorem-proving approach to database integrity", in *Foundations of Deductive Databases and Logic Programming*, J Minker (ed.), Morgan Kaufmann, Los Altos, CA, pp 313-362, 1988.

[9] Sommerville I, *Software Engineering (Fourth Edition)*, Addison Wesley, 1992.

[10] Carré B A, "Reliable Programming in Standard Languages", in *High Integrity Software*, C T Sennett (ed.), Pitman Publishing, pp. 102-121, 1989.

[11] Redmill F J (ed.), *Dependability of Critical Computer Systems 1*, Elsevier Applied Science, 1988.

[12] Cullyer W J, Goodenough S J and Wichman B A, "The Choice of Computer Languages for use in Safety-Critical Systems", in *Software Engineering Journal*, 1991.

[13] DeMillo R A, Lipton R J and Perlis A J, "Social Processes and Proofs of Theorems and Programs", in *CACM*, Vol. **12**, No, 5, 1979.

[14] Wichman B A, *Insecurities in the Ada Language*, National Physical Laboratory, Teddington, UK, Report DITC 137/89, 1989.

[15] Lampson B W, Horning J J, London R L, Mitchell J G and Popek G L, "Report on the Programming Language Euclid", in *ACM SIGPLAN Notices*, Vol. **12**, No. 2, 1977.

[16] Currie I F, "NewSpeak: a Reliable Programming Language", in *High Integrity Software*, C T Sennett (ed.), Pitman Publishing, 1989.

[17] Craigen D A, *Description of m-Verdi*, I P Sharp Technical Report TR-87-5420-02, 1987.

[18] Carré B A, Jennings T J, Maclennan F J and Farrow P F, *SPARK — The SPADE Ada Kernel*, Program Validation Limited, Southampton, UK, 1989.

[19] Carré B A and Debney C, *SPADE-Pascal*, Program Validation Limited, Southampton, UK, 1985.

[20] Clutterbuck D L, *SPADE-68020: Translation Strategy Document*, Program Validation Limited, Southampton, UK, 1989.

[21] Constantine L L and Yourdon E, *Structured Design*, Englewood Cliffs NJ: Prentice-Hall, 1979.

[22] Knight J C and Leveson N G, "An Experimental Evaluation of Independence in Multiversion Programming", in *Transactions on Software Engineering*, Vol. **SE-12**, No. 1, 1986.

[23] Polak W, *Compiler Specification and Verification*, Lecture Notes in Computer Science 124, Springer Verlag, 1981.

[24] Hennell M A, "Program analysis and systematic testing", in *High Integrity Software*, C T Sennett (ed.), Pitman Publishing, 1989.

[25] Parnas D L, van Schouwen J and Kwan S P, "Evaluation of Safety-critical Software", in *Communications of the ACM*, Vol. **3**, June 1990.\

[26] Redmill F (ed.), *Safety Critical Systems Club Newsletter*, Vol. 3, No. 1, 1993.

[27] *STARTS Purchases' Handbook*, NCC, Procuring Software-based Systems, 1989.

[28] ANSI/IEEE 729: *IEEE standard glossary of software engineering terminology*, IEEE, 1983.

[29] Lientz B and Swanson E B, *Software Maintenance Management*, Addison-Wesley, 1980.

[30] *The STARTS Guide: A guide to the methods and software tools for the construction of large real time systems* (Second Edition), NCC, Vols. 1-2, 1987.

[31] Woodcock J C P and Larsen P G (eds.), "Tool Descriptions", in *Formal Methods Europe Symposium (FME '93) : Industrial-Strength Formal Methods*, Lecture Notes in Computer Science 670, Springer-Verlag, pp. 679-89, 1993.

[32] O'Neill G and Parkin G I, *Draft Guidelines for the use of Formal Methods*, National Physical Laboratory, 1994.

[33] Woodcock J and Loomes M, *Software Engineering Mathematics*, Pitman Publishing, 1988.

[34] Cohen B, Harwood W T and Jackson M I, *The Specification of Complex Systems*, Addison-Wesley Publishing Company, 1986.

[35] Jones C B, *Systematic Software Development Using VDM*, Prentice Hall International, 1986.

[36] Spivey J M, *Understanding Z*, Cambridge University Press, Computer Science Tract 3, 1988.

[37] Spivey J M, *The Z Notation: A Reference Manual*, Prentice-Hall International, 1989.

[38] Goguen J A, "Parameterised Programming", in *IEEE Transactions on Software Engineering*, Vol. **SE-10**, pp. 528-544, 1984.

[39] Musser D R, "Abstract Data Type Specifications in the AFFIRM System", in *Proceedings of the Specification of Reliable Systems Conference*, Cambridge, Mass., USA, 1979.

[40] Gordon M, *A Proof Generating System for Higher-Order Logic*, University of Cambridge, Computer Laboratory, Technical Report 103, 1987.

[41] Hall A, "Seven Myths of Formal Methods", in *IEEE Transactions on Software Engineering*, 1990.

[42] Dale C (ed.), *Software Reliability and Metrics Club Newsletter*, Issue 11, 1993.

[43] McCabe T J, "A complexity measure", in *IEEE Transactions on Software Engineering*, Vol. **SE-2**, No. 4, 1976.

[44] Halstead M H, *Elements of software science*, Elsevier North-Holland, New York, 1977.

[45] Charniak E and McDermott D, *Introduction to Artificial Intelligence*, Addision Wesley, 1985.

[46] Redmill F J and Anderson T (eds.), *Safety-Critical Systems — Current Issues, Techniques and Standards*, Chapman and Hall, 1993.

[47] Adams J M and Taha A, "An Experiment in Software Redundancy with Diverse Methodologies", in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, 1992.

[48] Fagan M, "Design and code inspections to reduce errors in code development", in *IBM Systems Journal*, Vol. **15**, No. 3, 1976.

[49] Yourdon E, *Structured Walkthroughs*, Yourdon Press, New York, 1979.

Bibliography

ANSI/IEEE 1012, *Software Verification and Validation Plans*, IEEE, 1986.

AQAP 13: NATO Software Quality Control System Requirements, NATO International Staff — Defence Support Division, NATO: Allied Quality Assurance Publication, 1981.

Bowen J and Stavridou S, "Formal Methods: Epideictic or Apodeictic?", in *BCS FACS FACTS*, Series III, Vol. **1**, No. 3, 1993.

Bowen J and Stavridou S, "Safety-Critical Systems, Formal Methods and Standards", in *Software Engineering Journal*, 1993.

Bowen J and Stavridou S, "The Industrial Take-Up of Formal Methods in Safety-Critical and Other Areas: A Perspective", in *Formal Methods Europe Symposium (FME '93) : Industrial-Strength Formal Methods*, J C P Woodcock and P G Larsen (eds.), Lecture Notes in Computer Science 670, Springer-Verlag, 1993.

Brocklehurst S and Littlewood B, "New Ways to Get Accurate Reliability Measures", in *IEEE Software*, Vol. **9**, No. 4, July 1992.

Bromell J Y and Sadler S J, A "Strategy for the Development of Safety-Critical Software", in *Achieving Safety and Reliability with Computer Systems*, B K Daniels (ed.), Elsevier Applied Science, 1987

BS 4778, *Quality Vocabulary — Part 1: International Terms*, British Standards Institution, (Equivalent to ISO 8402), 1987.

BS 4778, *Quality Vocabulary — Part 2: Quality Concepts and Related Definitions*, British Standards Institution, (Equivalent to ISO 8402), 1991.

BS 5750 parts 1–6, *Quality Systems*, British Standards Institution, (Equivalent to ISO 9000), 1987.

Clutterbuck D L, "DRIVE REPORT : Review of Current Tools and Techniques for the Development of Safety-Critical Software", in *Software in Safety-Related Systems*, Wichmann B A (ed.), IEC/BCS Joint Study Report, Wiley, 1992.

Craigen D, Gerhart S and Ralston T J, "Formal Methods Reality Check: Industrial Usage", in *Formal Methods Europe Symposium (FME '93) : Industrial-Strength Formal Methods*, J C P Woodcock and P G Lars (eds.), Lecture Notes in Computer Science 670, Springer-Verlag, 1993.

DEFSTAN 00-55, *The Procurement of Safety-Critical Software in Defence Equipment*, Interim Defence Standard 00-55, Ministry of Defence, UK, 1991.

DEFSTAN 00-56, *Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment*, Interim Defence Standard 00-56, Ministry of Defence, UK, 1991.

DO178B, *Software Considerations in Airborne Systems and Equipment Certification*, Interim Draft 6.3, RTCA, 1992.

Drive Safely, *Towards a European Standard : The Development of Safe Road Transport Informatic Systems*, 1992.

Good D I, *Mathematical Forecasting Computational Logic Incorporated*, Technical Report No 47, 1989.

Hoare C A R, *ProCoS: Esprit Project BRA 3104*, Keynote Address, Z Refinement Workshop, IBM Hursley, UK, 1990.

Hobley I P, *Techniques for Enhancing the Reliability of Non-Critical Systems*, PhD Thesis, School of Computer Studies, University of Leeds, 1989.

IEC 880, *Software for Computers in the Safety Systems of Nuclear Power Stations*, Bureau Central de la Commission Electrotechnique Internationale, 1986.

IEC SC65A WG9 (Technical Committee No. 65: Industrial-Process Measurement and Control), *Draft — Software for Computers in the Application of Industrial Safety Related Systems*, International Electrotechnical Commission, 1989.

IEC SC65A WG10 (Technical Committee No. 65: Industrial-Process Measurement and Control), *Draft — Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, International Electrotechnical Commission, 1989.

Ince D C, *An Introduction to Discrete Mathematics and Formal System Specification*, Clarendon Press, 1988.

ISO 8879, *Information Processing — Text and Office Systems — Standard Generalised Markup Language (SGML).*

ISO 9000, *Quality Management and Quality Assurance Standards — Guidelines for Selection and Use*, International Standards Organisation, (equivalent to BS 5750), 1987.

ISO 9000-3, *Quality Management and Quality Assurance Standards — Part 3: Guide to the Application of ISO 9001 to the Development, Supply and Maintenance of Software*, (equivalent to BS 5750 Part 13), 1991.

Jackson D, "New Developments in Quality Management as a Prerequisite to Safety", in *Directions in Safety-Critical Systems: Proceedings of the Safety-Critical Systems Symposium, Bristol*, Redmill F and Anderson T (ed.), pp. 257–269, 1993.

JDS.02.03.06, *Code of Practice for Embedded Software*, Jaguar Cars Ltd, Draft 1, Jan. 1991.

Karger B, "Aspects of Proving Compiler Correctness", in *SAFECOMP'90*, Gatwick, UK, 1990.

Leveson N, "Safety as a Software Quality", in *IEEE Software*, Vol. **6**, No. 3, May 1989.

Leveson N G, "The Challenge of Building Process-Control Software", in *IEEE Software*, Vol. **7**, No. 6, Nov. 1990.

Littlewood B, "Modelling Growth in Software Reliability", in *Software Reliability Handbook*, ed. P Rook, Elsevier Applied Science, 1990.

MIL-STD-882B, *System Safety Program Requirements*, 1984.
Musa J D and Everett W W, "Software-Reliability Engineering: Technology for the 1990s", in *IEEE Software*, Vol. **7**, No. 6, pp. 36-43, 1990.

Proving the Correctness of Compiler Generated Object Code, Program Validation Limited, Technical Report, 1990.

Pygott C H, *Formal Proof of Correspondence between the Specification of a Hardware Module and its Gate Level Implementation*, Report No 85012, RSRE, Malvern, 1985.

Robinson A S, "A User Oriented Perspective of Fault Tolerant System Models and Terminology's", in *The 12th Annual International Conference on Fault Tolerant Computing*, pp. 22-28, June 1982.

SI 2932, *The Electromagnetic Compatibility Regulations*, UK Statutory Instrument, 1992.
Siewiorek D P and Swarz R S, *Reliable Computer Systems — Design and Evaluation*, Second Edition, Digital Press, 1992.

Wichmann B A (ed.), *Software in Safety-Related Systems*, IEC/BCS Joint Study Report, Wiley, 1992.

Tools Index

Here we provided addresses of suppliers of those tools referenced in this report. This list is based upon [30], [31] and [32].

B-TOOLKIT: B-Core (UK) Ltd, Magdalen Centre, The Oxford Science Park, Oxford OX4 4GA.
CADIZ: York Software Engineering Limited, University Of York, York YO1 5DD.
CORE: System Designers PLC, Pembroke House, Pembroke Broadway, Camberley, Surrey GU15 3XD.
DST-FUZZ: Deutsche System Technik, Edisonstr. 3, D-2300 Kiel, Germany.
FUZZ: Spivey J M, Computing Science Consultancy, 34 Westlands Grove, Stockton Lane, York YO2 0EF.
GENESIS: Imperial Software Technology, Cambridge, UK.
IFAD VDM-SL TOOLBOX: IFAD, Forskerparken 10, DK-5230 Odense M, Denmark.
MALPAS: TA Consultancy Services Limited, "Newnhams", West Street, Farnham, Surrey GU9 7EQ.
MURAL: C B Jones, Department of Computer Science, University of Manchester, Oxford Road, Manchester, UK.
PROOFPOWER: ICL Secure Systems, Eskdale Road, Winnersh, Wokingham, Berkshire RG11 5TT.
RAISE: CRI A/S, Bregnerødvej 144, DK-3460 Birkerøed, Denmark.
SPADE: Program Validation Limited, 26 Queen’s Terrace, Southampton SO1 1BQ.
SPECBOX: Adelard, 28 Rhondda Grove, London, UK.
VDM TOOL: Imperial Software Technology, 3 Glisson Road, Cambridge CB1 2HA.
Z TOOL: Imperial Software Technology, 3 Glisson Road, Cambridge CB1 2HA.

Appendix A — Rapid prototyping

Responsibility: *Project Manager and Software Manager*

A particularly difficult task in any system development is to ensure that the statement of requirements is adequate and that it has been correctly represented by the specification. Very often discrepancies only appear late on in the development, when some (or all) of the implementation has been completed. Peer review is clearly essential for the assessment of the statement of requirements, and for the Preliminary Hazard Analysis (see Appendix D). To help with the examination of the specification, it may be beneficial to model it using a technique known as prototyping.

A prototyped specification helps the procurer and the supplier to assess the adequacy and correctness of the specification. It can also be used to help with safety assessment, by experimentation with various input conditions. A prototype may be maintained throughout the development to evaluate proposed changes, for example.
Prototyped software is often implemented in functional or logic programming languages, such as Prolog, which allow for very rapid development. These languages are not in general efficient enough in execution to be used in the final implementation (and do not satisfy bounded memory constraints for safety-critical systems). This must be regarded as a good thing, for continued iteration of a prototype to a final solution is very difficult to control. **There should be no attempt to modify the prototype for actual service**.

A prototype is difficult to construct without first undertaking significant system design work. This may undermine its viability.

Appendix B — Example documentation

Responsibility: *All*

B.1 Software design plan
- *B.1.1 Integrity level*
- *B.1.2 Usage of formal methods*
- *B.1.3 Software architecture*
- *B.1.4 Software development techniques*
- *B.1.5 Testing strategy*
- *B.1.6 Software integration plan*

B.2 Software validation plan
- *B.2.1 Test plan*
  - B.2.1.1 Required input data
  - B.2.1.2 Expected output data
- *B.2.2 Acceptance criteria*
- *B.2.3 Operational environment*
- *B.2.4 Failure modes*
- *B.2.5 Safety functions*
  - B.2.5.1 Signal range
  - B.2.5.2 Logic combinations
  - B.2.5.3 Timing requirements
  - B.2.5.4 Response times
  - B.2.5.5 User displays (topic not covered within the MISRA Guidelines)
  - B.2.5.6 Output sequences
  - B.2.5.7 Failure recovery
  - B.2.5.8 Static tests
- *B.2.6 Evaluation of the results*
- *B.2.7 Considerations of previous operation*

B.3 Software design review plan
- *B.3.1 Techniques to be used*
- *B.3.2 Acceptance criteria*

B.4 Software acceptance plan
- *B.4.1 Acceptance process*
- *B.4.2 Acceptance criteria*
- *B.4.3 Sub-contract issues*

B.5 Software verification specification
- *B.5.1 Current stage in life-cycle*
- *B.5.2 Verification criteria*
- *B.5.3 Verification techniques*
- *B.5.4 Verification tools*
- *B.5.5 Verification activities*
  - B.5.5.1 System functions
  - B.5.5.2 System reliability
  - B.5.5.3 System performance
  - B.5.5.4 System safety requirements
- *B.5.6 Level of detail*

B.5.7 Independence

B.6 Software requirements specification

B.6.1 Software functional requirements

B.6.2 Software safety requirements

B.6.2.1 Software functional safety requirements

B.6.2.2 Software safety integrity requirements

B.6.3 Software non-functional requirements

B.6.3.1 Interfaces

B.6.3.2 Hardware/software constraints

B.6.3.3 Operating conditions

B.6.3.4 Commissioning

B.6.3.5 Repair

B.6.3.6 Maintenance

B.7 Software requirements verification results

B.8 Software detailed design specification

B.8.1 Description of software structure

The following aspects may be included:

- Integrity level
- Software size and complexity
- Software features
  - Abstraction
  - Modularity
  - Functionality
  - Information flow
  - Sequencing
  - Concurrency
  - Data structures
  - Human factors
  - Verification and validation
  - Software maintenance
  - Information hiding
  - Encapsulation
  - Self monitoring
  - Modular decomposition

B.9 Software module specification

For each module the following aspects may be included:

- Module identification
  - Specification
  - Interfaces
  - Design strategy
  - Module structure
  - "Off the shelf" usage
  - Module proof

B.10 Software verification results

B.10.1 Software design verification results

B.11 Software module design verification results

B.12 Software design review results

B.13 Software code and supporting documentation
B.13.1 Source code listings
B.13.1.1 Supporting documents
B.14 Code review results
B.14.1 Code reviewer's comments
B.14.2 Actions taken
B.15 Software acceptance test specification
B.15.1 Acceptance procedure
B.15.1.1 In-house software
B.15.1.2 Sub-contracted software
B.15.1.3 Acceptance criteria
B.15.1.4 Code
B.15.1.5 Tests
B.15.1.6 Documentation
B.16 Software module test specification
B.16.1 Software module tests
B.16.1.1 Execution of intended functions
B.16.1.1.1 Required input data
B.16.1.1.2 Expected output data
B.16.1.2 Non-execution of unintended functions
B.16.1.2.1 Required input data
B.16.1.2.2 Expected output data
B.17 Software integration test specification
B.17.1 System tests
B.17.1.1 Module interactions
B.17.1.1.1 Required input data
B.17.1.1.2 Expected output data
B.17.1.2 Execution of intended functions
B.17.1.2.1 Required input data
B.17.1.2.2 Expected output data
B.17.1.3 Non-execution of unintended functions
B.17.1.3.1 Required input data
B.17.1.3.2 Expected output data
B.17.1.4 Controlling hardware failures
B.17.1.4.1 Required input data
B.17.1.4.2 Expected output data
B.17.2 Software designers, implementers and test planners independence
B.18 Software module test results
B.18.1 Software version numbers
B.18.2 Static tests
B.18.3 Dynamic tests
B.18.4 Conformance to acceptance criteria
B.18.5 Test coverage
B.19 Software module error incidence results
B.19.1 Error characteristics
B.19.2 Actions
B.20 Software integration test results

B.20.1 Hardware configuration
B.20.2 Hardware version numbers
B.20.3 Software configuration
B.20.4 Software version numbers
B.20.5 Timing data
B.20.6 Event sequencing
B.20.7 Conformance to acceptance criteria

B.21 Software integration error incidence results
B.21.1 Error characteristics
B.21.2 Actions

B.22 Software validation results
B.22.1 Hardware configuration
B.22.2 Software configuration
B.22.3 Other equipment used
B.22.3.1 Calibration
B.22.4 Tools and simulation models used
B.22.5 Discrepancies
B.22.5.1 Actions taken
B.22.6 Validation
B.22.7 System compliance with requirements
B.22.8 Validation simulations

B.23 Software acceptance test results
B.23.1 Statements
B.23.2 Agreement (signatures)

B.24 Maintenance procedures

B.25 Quality management plan
B.25.1 Allocation of responsibilities
B.25.2 Quality system
B.25.3 Internal quality system audits
B.25.4 Corrective action

B.26 Quality assurance plan
B.26.1 Life-cycle activities
B.26.1.1 Contract review
B.26.1.2 Purchaser's requirements specification
B.26.1.3 Development planning
B.26.1.4 Quality planning
B.26.1.5 Design and implementation
B.26.1.6 Testing and validation
B.26.1.7 Acceptance
B.26.1.8 Replication, delivery and installation
B.26.1.9 Maintenance
B.26.2 Supporting activities
B.26.2.1 Configuration management
B.26.2.2 Document control
B.26.2.3 Quality records
B.26.2.4 Measurement
B.26.2.5 Rules, practices and conventions

Appendix C — Constraint analysis

C.1 Functional constraints

Responsibility: *Project Manager and Project Engineer*

When implementing a system care must be taken not only to adhere to the functional requirements but to also observe obvious constraints concerning the system behaviour. Such functional constraints typically include performance, efficiency, safety, security, reliability, quality and dependability. There is however a difficulty in separating out the functional requirements and the functional constraints, often due to ambiguities used in their descriptions.

The main difference between a functional requirement and a functional constraint is that the former describes the possible operations of the system whilst the latter provides information concerning when and how such operations may be carried out.

Thus, for a route guidance system, a functional requirement might be that an optimum route, from the current vehicles location to the desired destination, should be selected and displayed, whilst a functional constraint would be that the time taken to make this evaluation is less than the time for the vehicle's current location to change significantly.

C.2 Design constraints

Responsibility: *Project Manager and Project Engineer*

When requesting a system, the customer will in general provide the client with not only the required functionality of the system but also a number of extra conditions such as, *inter alia*, hardware and software standards, particular libraries and operating systems that must be used, and compatibility issues. These conditions affect the decisions that will have to be made during the design and development of the system.

Care must be taken to restrict the extent to which the customer specifies design constraints, which would otherwise inhibit the creativity associated with the implementation process. Similarly attention should be made to recognise whether over-specification has been made due to an inability to identify non-functional requirements. For example, the insistence on the use of assembler is typically a substitute for constraints on performance or efficiency. Thus care is needed in analysing design constraints to see if they are essential, unnecessary, or necessary but in the wrong form.

Appendix D — Software hazard analysis

D.1 A statement of the problem

Responsibility: *Software Manager and Software Engineer*

A hazard is a physical situation with a potential for an accident to occur (e.g. the failure of the brakes to operate when the pedal is depressed). Risk is a function of the likelihood of a hazard occurring, the likelihood the hazard will lead to an accident, and the severity of any potential accidents resulting from the hazard. Risk is reduced by reducing any or all of these factors.

Standard system safety analysis techniques rely upon historical information about the reliability of the individual components, and models that define the connection between these components, or of data about similar systems. However for software such historical information is normally not available since software is usually specifically constructed for each use, and random wear-out failures are not the problem.

Devising probabilistic models of software reliability is an important research topic — they are potentially very useful in software development, however their usefulness in validating safety is less clear. Although some meaningful failure probabilities can be obtained that may be applicable for fairly modest reliability levels, the very low failure probabilities required for high integrity safety-critical systems need more observation of the software than could possibly be obtained in a realistic amount of time.

Software reliability prediction models assume that it is possible to accurately predict the environment in which the software will be used, and to anticipate and specify correctly the appropriate behaviour of the software under all possible circumstances. Unfortunately failures most often occur as a result of flawed specifications due to faulty assumptions about the behaviour of the environment, or the required behaviour of the software.

In addition, in some cases the introduction of the computer itself causes a change in the environment (e.g. the behaviour of the user may change over time as a result of interacting with the computer).

D.2 Objectives

Responsibility: *Software Manager and Software Engineer*

The goal of software hazard analysis is to identify and categorise software-related hazards by likelihood and potential severity. For complex systems, it may be impossible to guarantee that all hazards have been identified and correctly assessed, however it is possible to make the system safer by optimising the design according to the hazard assessment and then planning contingency actions in case a mistake has been made.

Whilst determining all the causes of a hazardous state is much more difficult than identifying the hazard itself, fortunately this is not required to make a system acceptably safe. At worst, many systems can be designed to detect that a hazardous condition exists (without knowing why or how it occurred) and to take a protective action, such as failing safe. At best, analysis can identify and prevent some potential causes, thus eliminating the need to fail safe in those cases, and increasing the overall reliability and effectiveness of the system.

D.3 Techniques

Responsibility: *Software Manager and Software Engineer*

Once system hazards have been identified, fault-tree analysis or other modelling and analysis techniques can help to identify the software hazards, i.e. those actions or inactions of the software that alone, or with other events, can lead to the identified system hazards. These procedures involve the integration of system engineering and software engineering.

The identified software hazards can be used to write software safety requirements (or constraints), which are then included in the *Software Requirements Specification*. The safety requirements usually involve ways to eliminate and control hazards and to limit damage in the case of an accident. Whilst functional or mission requirements often focus on what the system shall do, safety requirements must also include what the system shall not do. An important part of the safety requirements is the specification of acceptable failure modes for the software.

The input to the software hazard analysis should be:

- the preliminary system hazards list
- the software architecture.

and its output should be:

- the software hazards list
- the detailed software safety design criteria
- the high-level test requirements.

To perform a hazard analysis on a safety-critical software component the following should be done:

a) analyze:
   - safety-critical software components for correctness and completeness, for input and output timings, for multiple events, for an out-of-sequence event, for the failure of an event, for an adverse environment, for deadlocking, for an incorrect event, for data with an inappropriate magnitude, for hardware failure sensitivities, etc.
   - software implementation of the safety requirements
   - possible combinations of hardware failures, software failures, transient errors,

and other events that could cause the system to operate in a hazardous manner
- proper error default handling for special characters or inappropriate or unexpected data in the input data stream
- fail-safe and fail-soft modes
- input overload or out-of-bound conditions.

b) Perform a static analysis:
- perform an internal path and control process flow analysis on safety-critical computer software components.

c) Propose change recommendations:
- recommend changes to the design, programming, and testing.

d) Support reviews:
- support reviews of each safety-critical software component.

Appendix E — Specification languages

E.1 Structured techniques

Responsibility: *Software Manager and Software Engineer*

Techniques which reduce the potential for ambiguity and support some degree of consistency checking, but which are not supported by a rigorous mathematical semantic definition are often called structured or semi-formal techniques. They may use textual or graphical notations, or both.

Advanced tools providing support for requirements analysis and specification, and favoured in the UK and Germany respectively are CORE and EPOS, both of which are described in the STARTS Guide [27].

In the CORE method eleven steps are defined to help gather relevant information, to propose data and process relationships, and to check the consistency of these relationships. An abstract requirement is subdivided into more manageable viewpoints (displayed graphically or textually), which are in turn subdivided, resulting in a structured hierarchical definition. This process can be continued down to the level of detailed design although other design-specific tools and methods may be more appropriate for this. CORE can help trace each requirement through to the design, and it includes a data dictionary which supports consistency checking. EPOS is similar, supporting a wide range of life-cycle activities.
SSADM, a widely used UK government standard, was designed primarily for commercial applications but it is believed by some to be of wider relevance. SSADM is based on entity-relation modelling techniques, and data flow diagrams.

Other methods, such as Jackson structure charts, Yourdon data flow diagrams and structured languages (which aim to restrict the power of expression provided by a natural language) are also used. However, these methods are probably more appropriate for lower levels of "specification" i.e. once a system has been decomposed into smaller subsystems, and other design work has been carried out.

These semi-formal techniques are clearly applicable to the description of functional requirements, but are not well-suited to the definition of non-functional requirements. Also, whilst diagrams may appear easier to understand, diagrammatic methods can produce very bulky specifications, and must be well structured to support any assessment of their completeness. An underlying database which allows one to trace the requirements through into the lower level specification and design documentation, will help significantly both development and assessment.

An alternative to the *Software Functional Requirements* is object-oriented design in which the system is viewed as a collection of interacting objects rather than as functions. The system state is decentralised, each object managing its own state information, and communicating with other objects via messages rather than sharing variables. As the objects are independent re-use is natural, and the developed system should be easier to maintain. However, whilst for

some classes of system there is a clear mapping between real world entities and their controlling objects, for others the identification of appropriate system objects can be difficult. It is not necessary to use an object-oriented language to implement an object-oriented design, although it is obvious easier to do so if language support for objects are available (see Appendix K).

To associated semantics with a specification additional, more precise, notation is required, for example to specify the safety-invariants of the system, or the control laws in use. This inevitably leads towards the use of formal methods for specification.

E.2 Formal methods

Responsibility: *Software Manager and Software Engineer*

Formal methods are currently being treated with caution by many companies, and it will be some time before they are widely accepted. Early users must therefore contend with the fact that not everyone will understand their use, though the requirement to always have a natural language description alongside the formal specification will go some way to alleviate the problem (see Appendix I).

Appendix F — Notes on the software functional requirements

Responsibility: *Project Engineer and Software Engineer*

Whilst most of the *Software Functional Requirements* will be found from the *System Requirements Specification*, there may also be other requirements which are not initially expressed explicitly. The following notes are intended to assist in the identification of both types of requirements so that the resulting specification can be made as complete and unambiguous as possible.

1) The *Software Functional Safety Requirements* form part of the *Software Functional Requirements*.
2) The *Software Requirements Specification* may contain aspects of minor importance to the software design but which provide a background to licensing, such as recommendations for functions or engineered safety features.
3) The *Software Functional Requirements* represent an expansion of the functions which are assigned to the sub-system and which are to be implemented by the sub-system.
4) Design constraints can be imposed by the requirement to adhere to other standards or regulations.
5) The safety requirements shall ensure that the system does not reach a hazardous or unsafe state, where an event may cause an accident. Moreover it shall be transparent from the safety requirements what actions should be taken, if an unforeseen event in the environment leads to an unsafe state.
6) The functional requirements shall be the properties of the product, not the project. They shall describe what has to be done and not how it has to be done.
7) The use of a formal specification technique or language may help to show coherence and completeness of the *Software Functional Requirements*. If the formal specification is only readable by persons with special training then a more readable interpretation shall be provided as well (e.g. a formal specification can have a natural language commentary along side). Computer assisted tools may be used to assist the writing of the specification.
8) The overall purpose of the software should be stated in terms of its effect on the equipment it aims to control. In so far as is practicable, this shall also include a description of what the software should not do. The relationship between these objectives and existing policies and traditional practices should also be indicated.
9) The functional requirements should be unequivocal, testable or verifiable, and feasible.
10) Constraints or requirements on the use of specific technical solutions should be listed.
11) The constraints between hardware and software shall be described with reference to the *System Requirements Specification*.
12) There are a number of attributes that can place specific requirements on the software, e.g.:

- reliability
- availability
- maintainability
- security
- transferability/conversion.

It is important that the required attributes are specified so that their achievement can be

objectively verified by a prescribed method.
13) The performance goals of the system — both intrinsic (such as throughput and accuracy) and extrinsic (such as economic) — should be stated. These should be ranked by priority where appropriate. For each one there should be a statement as to whether it is absolutely necessary for the software to be acceptable for its purpose, or whether it is a goal to be aimed for.
14) The function of the software should be stated in terms of the transformations between its inputs and outputs. The correspondence between these and the stated objectives and constraints should be indicated. The inputs and outputs should be identified explicitly and a preliminary description of them given. Software functions should be described in behavioural terms rather than in procedural ones.
15) The above preliminary functional requirements should then be expanded to produce the more detailed aspects of functional requirements.
16) Input and output handling functions should be derived from the requirements together with any necessary validation functions (such as format, field or logic checks).
17) Any necessary data base maintenance, updating and information retrieval functions should be specified.
18) Related data elements should be grouped and the group activity relationship (between each data group and the inputs and outputs of the software) determined, identifying any groups with special requirements.
19) All the application-specific calculations or application-dependent data manipulations that must be carried out should be fully specified, preferably with the help of diagrams depicting functional and I/O relationships.
20) The interface between the sub-system under consideration and any other sub-system, either within or outside the product, whether a direct connection exists or is planned, shall be identified and documented indicating the specific interfaces and the related *Software Requirements Specification*. Particular consideration should be given to:

- user interfaces
- hardware interfaces
- software interfaces
- communications interfaces.

21) Special operating conditions such as during commissioning, repair and maintenance must be included in the requirements.
22) Consideration must be given to all areas of failure and error handling, both from a managerial viewpoint, and in terms of what should be built into the software sub-system. Requirements should typically be specified for:

- software failure
- recovery procedures
- information collection
- information processing
- human error or violations.

23) Any requirements on external equipment and procedures which are necessitated by the introduction of the software should be specified. The requirements for these changes should be kept well separated from the "real" requirements.
24) Consideration should be given to those physical factors which may have no apparent relevance to the function of the software system, but which potentially influence the function nevertheless. These typically include the environmental factors which the software system

must be designed to tolerate.
25) Any technical terms, or common terms used with a special meaning should be defined.
26) The *Software Functional Requirements* documentation will be the definitive source of information for the software design process, for the development of the *Software Module Test Plan*, and for ongoing software maintenance and modification tasks. Therefore it is essential that an adequate level of formality is achieved.

Appendix G — Notes on the software safety integrity requirements

Responsibility: *Project Engineer and Software Engineer*

The formulation of the *Software Safety Integrity Requirements* ultimately determines the level of safety that is achieved. (It assumes that the *Software Requirements Specification* is correct.)

Safety integrity relates to the performance of the safety-related systems from a safety viewpoint, and can be specified for the total configuration of the safety-related system or for an individual safety-related sub-system.

It should be noted that whilst safety usually requires reliability, the converse is not necessarily true. The following is a list of features that should be considered in the design of a safety-critical system.

1) The software reliability requirements represent an expansion of the reliability requirements of the system.
2) It has been argued that faults in a complex system are a natural result of the complexity and coupling in a system. Unless great care is taken not to increase these factors when introducing computers into critical systems, the risk of faults may increase.
3) The software should monitor both itself and the hardware. This is considered a primary factor in the achievement of the overall system reliability requirements.
4) The software should be able to collect automatically all the information required during periodic testing.
5) Self monitoring shall meet the following requirements unless it is shown that other means provide the same objectives:

- those parts of the memory that contain code or invariable data shall be monitored to detect any changes
- the process should be monitored for logical correctness of function.

6) The software should be designed in such a manner that all essential functions detailed in the *Software Safety Requirements* shall be testable during the operation of the system.
7) If any failure is detected during the operation of the system, appropriate and timely automatic actions shall be taken. This may require giving due consideration to avoiding spurious failures.
8) System self-checking shall be designed so as to meet the requirements of periodic testing. There shall be periodic testing to ensure that:

- every single function is covered by periodic testing, where possible
- any degradation of the execution of safety functions shall be detected
- the basic self-checking functions shall be tested during periodic tests.

9) One way to obtain an indication of the current reliability of the software is to keep a note of the number of faults discovered in the software during its life time from the beginning of the development life-cycle. Subject to statistical noise, a downward trend should indicate an increase in reliability. This can be particularly useful for large programs.

Appendix H — Animation

Responsibility: *Project Manager and Software Engineer*

Animation involves "executing" a given *Software Requirements Specification*. Thus we require that the requirements are either written in an executable specification language or that there is an animator capable of animating the language used to describe the requirements. Unfortunately this may lead to the use of a particular specification notation that, whilst lending itself to the animation process, is less suited to the application in question.

The main advantage of animation over rapid prototyping (see Appendix A), is that whilst a prototype is an actual implementation, an animation of a design uses only the requirements details and thus doesn't require any implementation decisions being made. For this reason the final system is more likely to agree with the results of the animation process than with an early prototype.

However, since the animation process is requirements based, it is wholly dependant upon the detail supplied by the *Software Requirements Specification* and will be of less use if these requirements are abstract. Similarly, animation makes no use of any information provided by the constraint analysis (see Appendix C) process, and thus may make a non-viable design appear attractive.

Appendix I — Formal methods

A formal method is a description, based on a self-consistent mathematical theory of axioms and rules of inference, which is amenable to objective analysis, manipulation and proof. Formal methods involve the use of formal logic, discrete mathematics, and machine-readable languages for the specification and verification of software. Formal development consists of a finite sequence of formal models, starting with the *System Requirements Specification* and ending at the machine level of the running system, each of which (except the first) is derived from the previous one and verifiable against it. The effects of such analysis include: substantiation that the specification is met, identification of which parts of the code or specification/architecture meet the next higher level of specification, and justifications as to why that specification is met. Developing software using these methods can lead to an implementation whose operational behaviour is known, with high confidence, to fall within a well-defined envelope. However, experience has shown that it is difficult to train users and purchasers to read or write formal specifications unless they are mathematically inclined. There are ways of overcoming this problem, for example, via animation, but formal methods are a relatively new technology and do not yet enjoy a widespread application. Even so, there is a growing interest in them, particularly in the area of safety-critical and high-integrity software. As more software engineers are trained in this discipline, the tool support will inevitably improve, broadening the application of formal methods still further. The use of a structured method to decompose a system from its formal specification into detailed design may assist the "rigorous" approach, although the correctness of the refinements must still be demonstrated.

For clear presentation each formal specification should be embedded within a natural language commentary. The formal specification, however, should be the definitive text.

I.1 Objectives of formal methods

Responsibility: *Software Engineer*

The goal of applying formal methods is to prevent and eliminate specification, design and implementation errors throughout software development. Thus, formal methods are complementary to testing. Whilst testing is used to show that functional requirements are met, formal methods could be used to increase confidence that anomalous behaviour will not occur for inputs which are out of range, or unlikely to occur.

A number of formal specification languages based on mathematical theories have now been developed to a stage so that many can be considered for use in specifying the functional requirements of practical systems. They have a number of advantages:

- they provide a means of stating unambiguously and precisely what is intended. Software engineering tools are now available for some of these languages that will check for ambiguities
- software engineering tools are available for some languages that enable the specification to be "animated", or tested

- their use "concentrates the mind wonderfully" on precisely what it is the system is intended to do, enabling many faults to be found at an early stage in the development life-cycle
- they can provide information in a form that is suitable for tasks that should be performed later on in the life-cycle (e.g. static analysis (see Appendix J)).

The benefits of formality include:

- as a way of thought, it promotes good engineering practice throughout the development cycle, leading to more understandable systems
- it facilitates prototyping and early validation of the *System Requirements Specification*, thus obviating the costs associated with developing an unacceptable system and having to modify this very much later
- it facilitates analysis and the prediction of properties of the system being developed in an objective and repeatable way
- it highlights those areas of the design where defensive measures should be included to assure correctness of system operation
- it provides support for dynamic testing
- it offers the potential of automating the analysis, manipulation and proof of the design and specification
- it offers the potential for absolute certainty in the verification process.

I.2 Criteria for formal methods

Responsibility: *Software Engineer*

In order to be suitable for safety-critical applications, a formal method should meet the following criteria:

a) It should contain a formal notation for describing specifications and designs in a mathematically precise manner. This notation should have a formally defined syntax and a semantics in terms of generally understood mathematical concepts.
b) It should have a proof theory for the verification of design steps.
c) It should provide guidance on good strategies for building a verifiable design.
d) It should be accessible in the public domain.
e) Case studies published in the open literature should demonstrate its successful industrial use.
f) It should be suitable for design as well as specification, either on its own or in combination with another formal method.
g) Courses and textbooks should be available.
h) A recognised standard version should exist. This should preferably be an international or national standard or published draft standard.
i) It should be supported by industrialised tools.

I.3 Use of formal methods

Responsibility: *Software Engineer*

The following are considerations regarding the degree of formal methods application:

a) *Levels of the design structure*: The most thorough use of formal methods begins by formally specifying high-level *Software Requirements Specification* and then formally verifying that they meet important system requirements (usually constraints on acceptable operation). The next level of requirements are then shown to meet the high-level requirements. Continuing this process down to the implementation would yield a proof that the implemented software meets the *System Requirements Specification*. Application of formal methods can start and stop at any consecutive level of the design structure, yielding proof that important requirements are captured down through those levels.

b) *Coverage of the System Requirements Specification and the Software Requirements Specification*: There are a wide range of formal methods available and each has its own specific application area. Thus once the design requirements have been decided upon a suitable formal method might be chosen for the attributes that it emphasises. The types of requirements that should be considered for the choice of applicability of a particular formal method are those that:
   1) Are most important with respect to system safety.
   2) Can be formalised within the domain of discrete mathematics.
   3) Involve complex behaviour, such as, concurrency, distributed processing, redundancy management, and synchronisation.

   These criteria could be used to select the set of requirements at a particular level of the design structure to which formal methods are applied.

c) *Degree of rigour*: Formal methods include the following three increasingly rigorous levels of activities:
   1) Formal specifications.
   2) Formal specifications with manual proofs.
   3) Formal specifications with automatically checked or generated proofs.

The use of formal methods alone forces specifications to be unambiguous. Manual proof is a well-understood process that can be used when there are small amounts of detail. Automatically checked or generated proofs assist the human proof process and offer a higher degree of dependability, especially in more complicated proofs.
The above criteria should therefore be used to plan the application of formal methods to software development. On occasions it might be possible to use other techniques to achieve the same aim as the use of formal methods. In this case their equivalence must be justified, and a statement of the effects of these alternative techniques on the development process should be provided.

I.4 Available tools

Responsibility: *Software Engineer*

Computer systems can in general be satisfactorily modelled with discrete mathematics - set theory, propositional and predicate logic [33]. This mathematical base means that software engineers can, in line with other engineering disciplines, construct formal system models whose consistency can be demonstrated [34], and from which implementations can be rigorously derived.

Two formal specification languages favoured in the United Kingdom are Z and VDM. Both support a model-based approach to specification, in which a system model is constructed from abstract primitives. VDM [35] encourages a layered top-down development of systems. At the top level a specification is an abstract model which identifies the system states, and defines the system operations. Data objects are specified as abstract mathematical data types, such as sets and mappings, and the operations on these data objects may be functions, specified with pre- and post-conditions. Z [36, 37] is based on set theory and predicate logic, but has an additional abstraction mechanism, called a schema, to help structure a specification. Z is the subject of a great deal of research, particularly at the Programming Research Group in the University of Oxford, although at the moment VDM is more widely used in industry for embedded system specification. There is also progress towards a British Standards Institution VDM standard.

An alternative style of specification is property-based, in which a specification comprises axioms which define the relationships between the operations of the system. Algebraic specification languages such as OBJ [38] fall into this category. A strength of the model-based approach of Z and VDM is that the resulting specification can be more clearly related to the proposed system, and is therefore better suited to "what if?" analyses and comparison back to the original statement of requirements. Checking the consistency of the rules in an algebraic language is difficult, and the problem of determining whether an arbitrary set of rewrite rules (the constituents of an OBJ specification) is convergent, i.e. that they will terminate and the result will be independent of their order of application, is known to be algorithmically undecidable [39].

Other formal languages are in use (e.g. HOL [40]), many of which are under evaluation in various applied research projects. A good overview of formal methods can be found in [34] and [7].

Formal specification can be carried out at varying levels of abstraction, as appropriate to the complexity of the system being implemented. As the system is decomposed, further details will emerge and the formal specification may be extended. Indeed both VDM and Z have an associated set of rules and procedures which support refinement (or reification) of specifications towards implementations. Each such refinement can (in theory) be proven correct. This is done in VDM by showing retrospectively that the derived implementation satisfies necessary conditions. An approach proposed for Z is a "refinement calculus" where correctness of refinements is assumed by their method of construction. Either way the work involved in formally verifying the correctness of the refinement steps is very substantial, and is still a research topic. A less formal approach (sometimes called the rigorous approach - an intuitive argument which could be backed up by formal proof) is more usual. In fact it is convincingly argued [41] that formal methods are most effective when used for abstract specification without proof of refinement.

A good formal specification comprises precise mathematics with a supporting natural language description which should read clearly with the mathematics omitted. The procurer of the system can then easily read the document and may also review the mathematics. In addition, this "redundancy" provides a source of reference against which an external reviewer can assess the formal description.

Tool support for formal methods is developing and a number of syntax editors and type checkers now exist for VDM and Z. In addition theorem provers and proof assistants are being developed to discharge the proof obligations resulting from the specification and refinement work. Prototyping as a means of checking the specification is considered in Appendix A.

In addition there are some special purpose packages that provide an integrated environment for formal specification and subsequent static analysis of safety-critical software.

I.4.1 Training

Before developers can make effective use of any of the above tools and techniques they will need to undergo a considerable amount of training. It should also be noted that it is fairly difficult to obtain the fullest usage of any new tool or technique in isolation, and thus an organisation would be advised to train a formal methods group rather than an single expert.

A formal specification is easier to comprehended than it is to produce. Thus, whilst a formal specification is a useful medium for a developer to communicate with an untrained customer, it will require skill on the part of that developer to produce the formal specification accurately.

This requirement for thorough training is not a restriction because there are many commercial training courses on most of the well known tools and techniques.

I.5 Formal proof

Responsibility: *Software Engineer*

Whilst the use of high level abstractions (e.g. specification languages or high level programming languages) provide a means of handling complexity, their use does not guarantee the absence of design errors. Formal proofs have been proposed as a mechanism to solve this problem.

Formal proof is the use of a formal, mathematically based, method to express the software and hardware design and development and to verify conformity between the various stages of the life-cycle starting with the formal *Software Requirements Specification*. A Formal proof is a strictly well-formed sequence of logical formulae such that each one follows from formulae appearing earlier in the sequence, or as instances of axioms of the logical theory. Formal proofs are often highly intricate. They proceed by simple matching of syntactic structure and as such are strongly dependent upon the form of syntactic categories such as

formulae and terms.

Formal proofs attempt to demonstrate code correctness by restructuring the code as a mathematical proof. There are two basic approaches to formal proofs. The first approach creates a logical representation of the code and proceeds to prove theorems upon this secondary representation. One must ensure that the translation process from the code to the logical representation does not itself introduce errors. The second approach integrates the logical representation with the code. While proving the equivalence between two algorithms (such as a higher level specification and a lower level implementation) has been demonstrated, the proof of correctness of an arbitrary piece of code requires substantially more research.

The highest degree of assurance in the design can be obtained by providing all supporting proofs and checking them with a *proof checker* (see Appendix I.6.2). A proof checker can be a relatively simple program and thus can itself be verified by formal proof.

I.6 Formal argument

Responsibility: *Software Engineer*

Formal arguments are constructed using either formal proofs (see Appendix I.5) or rigorous arguments.

Creation of a formal proofs will, however, consume a considerable amount of the time of skilled staff. A lower level of design assurance can be obtained by a rigorous argument. A rigorous argument is not a formal proof and is no substitute for it. It is at the level of a mathematical argument in the scientific literature that will be subjected to peer review. It is a high-level sketch of how a formal proof could be constructed, indicating the general form of the deduction in terms of the main steps and any auxiliary lemmas that are required, whose details could be filled in if required. A rigorous argument also has a role in providing a commentary on intricate formal proof, to facilitate review by the independent auditor and the verification and validation team. The key criterion for the acceptability of a rigorous argument is that it can be converted to a formal proof by the addition of detail. Thus the specification must itself be written in a notation whose formal semantics is recorded, the design step must be recorded in a similarly supported language, and any proof obligation on whose correctness the step depends must be in a mathematical notation. The proof obligations necessary to show that the formal specification is internally consistent are discharged by means of formal arguments. If such a rigorous argument was called into question, it would be clear what further formal work would be required to settle the doubt.

Where animation of the formal specification is used to validate the *Software Requirements Specification* against the *System Requirements Specification* this animation should be carried out by the construction of formal arguments, showing that the formal specification embodies all the safety features described in the *System Requirements Specification*. Formal arguments should be employed to show the link between the formal specification and the executable prototype (Appendix A).

In summary, the practical options for formal arguments of verification, in decreasing level of assurance, include:

- machine generated formal proof, checked by an independent proof checker
- manually generated formal proof, checked by an independent tool,

rigorous argument, checked by review.

I.6.1 Review of formal arguments

The verification and validation team should establish that the design team has produced sufficient formal arguments to discharge all the necessary proof obligations (see Appendix I.5). The verification and validation team should scrutinise all rigorous arguments produced by the design team for correctness and completeness. If found to be unsatisfactory, they should be returned to the design team for elaboration or correction. the verification and validation team should check the formal proofs for correctness and completeness using, if possible, a diverse tool from any employed by the design team in the discovery of the proof.

I.6.2 Proof checkers

In practice, it is very unlikely that formal proofs of any size will be created by hand. Instead, they will be developed using theorem proving assistants, which are interactive programs that carry out symbol manipulation under the guidance of a human operator. But theorem proving assistants are large programs whose correctness cannot readily be demonstrated by formal proof. It is, however, possible to remove the reliance on the correctness of the theorem proving assistant from the case for correctness of an application by arranging that a version of the final proof (omitting all the history of its construction) is passed from the theorem proving assistant to a proof checker. For reasonable languages, such a proof checker could be a very simple program (perhaps ten pages in a functional programming language) that could be developed to the highest level of assurance.

The input to the proof checker would be the formal proofs to be checked. Ideally, a proof checker could be designed which coped with the output from a range of theorem proving assistants. Unfortunately, the sort of standardisation work on which this would depend is nowhere in hand.

In the short term it is therefore likely that a separate proof checker would be associated with each theorem proving assistant. Such a proof checker would be based on a formalised proof theory, the consistency of which would have to be established. The proof checker should be verified against this proof theory.

I.7 Risks and limitations of formal methods

Responsibility: *Project Manager and Software Manager*

Formal methods are no panacea: it is possible to construct a "poor" formal specification -

which is difficult to check for well-formation and functional consistency. Thus formal methods are not the complete answer to all safety issues and their efficacy is not yet quantified. At present, engineering judgement, based on considerations of criticality, size, cost and available expertise, is needed to decide whether or not they should be applied to particular systems, and a combined approach, mixing some formality with other well established structured techniques, is often seen as being effective. There is a need for an infrastructure of support tools and training to support formal methods before their full potential can be reaped. Nevertheless, they merit serious consideration; formal methods are advantageous in the software development process, both for their analytic power and their influence on the thought processes underlying the development.

The completeness of any specification (formal or otherwise) can never be fully established. However, formal methods offer many advantages. Their underlying mathematics helps to ensure that a specification is unambiguous, and can support well-formation and functional consistency checks. Perhaps most importantly, a formal specification will force early consideration of all relevant issues, and provide a sound basis for review. Formal methods should be used for systems requiring a high level of integrity [48].

Usage of formal methods can be at a number of different levels, the lowest providing little more than structured methods. Formal methods may simply be used for a high-level specification of the system to be designed (e.g., using the Z notation). They may be applied to the development process (e.g., VDM). Or more rigorously, the whole process of proof may be mechanized.

In the long run, it seems likely that formal methods will become more widely used for the *Software Requirements Specification*. However, before this happens they will probably evolve substantially; for example, to have much stronger links with structured methods. This evolution will require further research into a number of aspects, for example, the ability of formal methods to specify safety properties and perform verification and validation.

The National Physical Laboratory (NPL), under the instruction of the Department of Trade and Industry (DTI), has recently performed three surveys (literature, the academic community and industrial members) of the uptake of formal methods and more importantly the issues surrounding why this uptake has not followed the predicted trend [42]. The main aim of these surveys was to learn the views of people using, or considering using, formal methods in three areas; benefits, limitations, barriers. The three surveys highlighted different reasons for the unwillingness to make use of formal methods.

The literature survey highlighted the lack of tool support and the complexity of the mathematics involved. The academic survey revealed that many institutions were trying to redress the balance by teaching formal methods, especially Z and VDM-SL. The industrial survey showed that whilst over half of those surveyed used formal methods, of some sort, they generally only adopted their usage because of customer requirements (e.g. Ministry of Defence (MoD)), or because they realised that they were working with safety-critical software and therefore required best practice.

Where formal methods are being integrated into the software life-cycle, this is most widely

done through structured methods, and much less through the use of requirements analysis tools.

For those that chose not to use formal methods the following three principal reasons were given:

- there is a lack of tools for formal methods, in particular commercially supported tools. There could be several reasons for this: not a large enough market, lack of standards, or not clear what type of tools are needed
- it has not been shown conclusively that there are cost benefits to be gained from the use of formal methods in producing software
- many of the barriers (and limitations) to the use of formal methods are the symptoms of the process of change from the use of one technique to a completely different one. This process will take some time to work itself out.

The STARTS Purchasers' Handbook [27] adds to this list and also suggests that the following are additional constraints associated with the applicability of available formal methods:

- not all technical areas are finalised in current formal models, although there is a lot of research in this area, for example:
  - concurrency issues
  - numerical precision, particularly of floating real types
  - temporal and performance issues.
- they are time intensive and require a high level of expertise
- they are difficult to apply on large scale problems.

Thus before formal methods are fully adopted by industry a number of changes will have to occur, for example:

- a programme of education to spread the understanding of formal methods
- research on metrics and data collection to help in assessing the contribution of formal methods to the production of software
- case studies, which may show the cost benefits to be gained by using formal methods
- efforts to get formal methods, especially VDM and Z, standardised.

Appendix J — Static analysis

Responsibility: *Software Engineer*

Static analysis does not require the execution of the program being analysed. It can be used to determine properties of the program which are universally true for all possible execution conditions. Static analysis methods range in sophistication from code of practice audit and structural complexity analysis, to flow and semantic analysis, and formal verification.

Code of practice audit usually only addresses conformance to naming and layout conventions etc. It may however also encompass more sophisticated well-formation checks such as language restrictions, control structure, data usage, etc. For reliable application of these checks, tool support is required (see Appendix I).

Attempts have been made [43, 44] to define metrics for the measurement of code complexity. However, code complexity depends upon many factors and is difficult to define precisely. Consequently, such metrics are considered to be of little use in assessing the fitness for purpose of high-integrity software.

A more effective application for static analysis is the demonstration of a program's well-formation with respect to its control-, data- and information-flow, and of its functional consistency with its specification. Such analyses are performed for all paths through the program and for all input data over a defined range. The static characteristic of a unit must be compared with the specification, especially for completeness and consistency.
A typical static analysis system might comprise flow analysers, e.g. control, data and information flow, and semantic analysis tools, e.g. verification condition generator, proof checker and symbolic interpreter. A number of commercial tools are now available that, typically, work with a "safe subset" of a language with additional information being provided in a special form of comment statement.

The flow analysers check that a program is well-formed with respect to its control structure, data usage and information flow. If a program is found to be defective in any of these respects, the errors or anomalies are described in a flow analysis report. It is helpful if each flow analyser produces tabular output.

Typical problems detected by these tools are unstructured code; reference to undefined variables or the non-use of variable definitions; incorrect import/export specifications (dependency relations) for a module; stable loop-exit conditions and redundant statements. (Note that although redundant code should normally be removed, it may exist for a valid reason.) Ideally flow analysis is done as the code is being developed.

After checking the integrity of a program's control, data and information flow, one can then verify that the program performs the function required of it. This is achieved through the use of the semantic analysis tools. Such tools can generate both path functions and verification conditions.

A path function comprises a traversal condition and an action. The traversal condition for

a path states the conditions under which that path is executed; the action expresses the final values of all variables in terms of their values at the initial endpoint of the path. A path function is generated for each path through the program. This information is most often used for direct comparison with the specification document. For example, if the specification states that the windows must automatically close if the key of the driver's door is inserted, then we would expect at least one path condition of the program to contain a traversal condition of the form *DoorkeyInserted*, and we would then check that the action for such path contained the assignment *CloseWindows*. We would also check that no other untoward actions took place under this condition, and that the windows were not closed under any other unspecified condition.

Path function information can also be used to guide the generation of structural test data. By supplying the traversal conditions of each path through the program semantic analysis might identify all the input test values required for full path coverage of the program.

Appendix K — Object-orientation

Atkins and Brown [7] describe an object-oriented system as a space which contains many independent objects. Each object provides a behaviour, which is a set of operations that the object can be requested to carry out. An object's actions are carried out by internal computation and by requesting other objects, which it can name, to carry out operations in turn.

[9] defines an object-oriented approach as being based on entities (objects) which have a hidden state and operations on that state. The system is expressed in terms of services requested and provided by interacting objects.

The characteristics of an object-oriented approach are:

- once an object has been defined its interfaces should never be altered. If a different interface is required then a new object should be designed based on the previous object
- shared data areas are eliminated. Objects communicate by exchanging messages rather than sharing variables. This reduces overall system coupling as there is no possibility of unexpected modifications to shared information
- objects are independent entities that may readily be changed because all state and representation information is held within the object itself. No access and hence no deliberate or accidental use of this information by other objects is possible. Changes to the representation may be made without reference to other system objects
- objects may be distributed and may execute either sequentially or in parallel. Decisions on parallelism need not be taken at an early stage of the design process.

K.1 Object-orientation techniques

Responsibility: *Software Engineer*

Object-orientation has recently been addressed at all phases within the life-cycle. [7] suggests the following three techniques:

- Object-Oriented Requirements Analysis techniques have been developed which involve early identification of candidate objects and their relationships. The objects are organised into classes, have both state and operational aspects, and so on. The requirements definition consists of documentation describing these objects, together with (graphical) descriptions of their possible interrelationships
- object-oriented specification languages are an attempt to try to impose an object-oriented framework onto existing specification languages. For example, the object-oriented approach may be particularly useful as a structuring

framework for developing and browsing large specifications

- object-oriented design techniques have been under investigation for a number of years. As an extension of the Object-Oriented Requirements Analysis techniques described above, in object-oriented design methods the notion of an object is used as the primary focus for structuring information.

Clearly, it is not essential to use an object-oriented approach in all of the phases of software development. For example, an object-oriented system design need not necessarily be implemented in an object-oriented programming language. However, by doing so, a cleaner conceptual mapping between the design and programming phases of a software project is provided. Indeed, as it is the management of the relationships between the many system models which is a major factor in the success or failure of a large software project, particularly in the event of errors, omissions, and enhancements to those models, a consistent view of the full software life-cycle is particularly important.

K.1.1 Advantages

General advantages commonly claimed for the object-oriented approach include the ability to devise a model of a problem in terms of the data items and operators which are most suited to the designers needs. The internal details of the data and operators are ignored so that all one needs to know is *what* they do, not *how* they do it.

In addition, by allowing one object to be defined in terms of previously defined objects, a hierarchy is created in which objects are defined and manipulated at increasing levels of abstraction at each level in the hierarchy. The solution to a problem can then be defined at the most appropriate abstract level, using the objects defined at that level. This helps to control the complexity of applications, which is often increased by attempting to define a solution at an inappropriate level of abstraction.

Specific advantages [7] which arise from direct support for abstraction include:

- *Data hiding* — the internal state of an object is hidden from its users. As a result, the clients of an object need only concern themselves with the service the object provides, not the methods by which that service is provided. Thus, not only are clients of an object not distracted by such irrelevant details, they are also unable to attempt to access private parts of the server object's state.
- *Data independence* — by restricting the use of an object to a fixed set of operators, one can control the extent to which the client object relies on the internal details of the server object.
- *Modularity* — the definition of appropriate objects can act as the focal point for modularising the implementation of a large software system, often providing a boundary for distribution or concurrency of objects. The object-oriented approach naturally encourages designs involving a small number of relatively independent object types interacting in well-defined ways.
- *Re-use* — by organising the classes within a class hierarchy, common properties of a class can be filtered out and inherited from one class to the next. In this way common state variables and methods need not be

re-implemented, but are shared between classes. In addition, by identifying the main object classes within an application, the object-oriented approach lends itself to the creation of object libraries in which commonly used object classes are maintained.

K.2 Object-oriented design

Responsibility: *Software Engineer*

Object-oriented design differs from the more familiar functional approach to design in that it views a software system as a set of interacting objects, with their own private state, rather than as a set of functions, and is based on the idea of information hiding or abstraction. Information hiding is a design strategy in which as much information as possible is hidden within design components. The basic premise which underlies it is the notion that the binding of logical control and data structures to their realisations should be made as late as possible in the design process. This means that communication between design entities is minimised (thus increasing the understandability of the design) and that the design is relatively easy to change as a modification affects only a minimal number of entities.

The state of the object may not be accessed except via operations. Conceptually, objects communicate by passing messages to each other and these messages initiate object operations. Thus, communication may be asynchronous and an object-oriented design may be realised as a parallel or a sequential program. In practice, object operation instantiation is often implemented as function or procedure calls but this is an implementation rather than a design decision. As in other modular structures, a redesign of the internal data structures and access mechanisms would not impact the rest of the system.

Object-oriented design is a way of considering a software design and is not dependent on any specific implementation language. Features such as data encapsulation make an object-oriented design simpler to realise although, it is perfectly possible to implement an object-oriented design in languages such as Pascal which do not incorporate such features.

K.2.1 Object-oriented programming

Object-oriented design is often confused with object-oriented programming. Object-oriented programming languages support the notion of objects and allow an object-oriented design to be implemented directly. They also incorporate notions of inheritance and run-time binding of operations to objects with the result that they inevitably have a high run-time overhead. Although the hierarchical structure is important for handling the complexity of the design process one of the objections to object-oriented programming is this extra overhead required in making subroutine calls and returns every time a module boundary is crossed. For this reason, they are unsuitable for some kinds of system building and have not been widely used for large-scale software engineering. A preprocessor can, however, remove a major portion of this overhead by "flattening" the hierarchy into a smaller number of levels. Thus, object-oriented programs need not be substantially slower at run time than more classical programming styles.

K.3 Risks and limitations of object-orientation

Responsibility: *Software Engineer*

It is worth noting that while there has been a great deal of interest expressed in object-oriented design techniques, they are much less mature than other equivalent implementation techniques, and it still remains to be seen how effective they are in practice.

There is also an overhead involved with using an object-oriented programming language. There is a considerable amount of code required to control the individual objects and thus if a small program were to be coded in an object-oriented manner there would be a large proportion of the final object code which is not actually part of the problem solutions. This control code becomes less significant when the program becomes larger and so an object-oriented approach is recommended only for large problems. For a safety-related system this "hidden" code would have to have been developed to the same integrity level as the final application requires.

The developer will invariably have a functional view of the system and translating this into an object-oriented view can be difficult.

Appendix L — Artificial intelligence

For general information on Artificial Intelligence [45] is recommended.

L.1 Expert systems

Responsibility: *Software Engineer*

[46] defines an expert system as a computer system that gives advice in areas where human expertise would normally be required. An expert system embodies a well defined (and usually narrow) aspect of the knowledge of a (number of) human expert(s) in the particular subject and makes it available to the non-expert. Expert systems are developed in a similar way to prototypes (see Appendix A). An initial working model is produced and then modified in response to the comments of a human consultant. The only difference between the processes is that in the case of expert system construction, the person who is shown the system is the human consultant, while with prototyping the customer or their representative is shown the prototype.

Although the idea of using computers to advise on specialist decision making has been discussed for many years, it only achieved prominence with the introduction of "knowledge based" techniques from the field of Artificial Intelligence. Knowledge based systems emphasised:

- symbolic computation, as distinct from traditional numerical methods, with symbolic data structures used explicitly to represent the specialised knowledge of human experts
- declarative representations of knowledge (encoding what needs to be done while leaving to the computer the question of which algorithms to use)
- domain-specific heuristics, or rules of thumb, for problem-solving, rather than formalised information-processing techniques
- strict separation of the knowledge base from the shell program that applies the knowledge in a particular situation
- shell design based on one or more generalised tasks, such as diagnosis, event monitoring, planning, or design tasks.

Many expert systems are developed using expert system shells. A shell will consist of a knowledge representation language, an interface engine and user interface facilities, and usually include interfaces to other software packages or languages.

Software associated with expert system technology has excellent interactive development facilities and hence is suitable for prototyping. However the size of prototypes generated is rather limited, expert systems can incorporate at most several hundred rules.

Given the serious consequences of adopting unproven methods, the standard of acceptance of expert systems is surprisingly high. From this point of view, one of the great weaknesses

of expert systems technology has been the relatively ad hoc nature of applications development. The idea of using "rules of thumb" is hardly likely to inspire confidence, and the lack of a strong theoretical foundation for expert system development compares unfavourably with other branches of engineering. It is therefore not surprising if software engineers, prefer to stick with professional human "judgement" rather than adopt a technology which, for many, implies a surrendering of responsibility to a machine, particularly one that is poorly understood. However, to offset these fears, it must be noted that, unlike most programming languages, expert systems could be rigorously defined and their behaviour is predictable.

The routine adoption of expert systems will require many obstacles to be overcome. Making expert systems sound and safe will require the introduction of more formal design and development techniques.

How effective such techniques will be depends on several factors including:

- the quality of the underlying design
- the quality and applicability of the rules and knowledge already included in the tool
- the ability to customise the rules and knowledge to the customer's own needs.

L.2 Neural networks

Responsibility: *Software Engineer*

[46] defines a neural network as consisting of a number of simple processors, or neurons, linked together. The neurons combine their inputs and subsequently produce an output which is passed to other neurons. The links between neurons contain weights which control the amplitude of the signal passing through. In addition, each neuron has an associated bias. It is these weights and biases that embody the information required to classify the input signals, just as in the biological brain it is the links between the neurons that determine function. A neural network is essentially a self-learning system. In the training stage, the weights and biases are iteratively improved, from an initial random setup, by applying input and output pairs to the network.

There a two significant problems with neural networks:

- the quality of the result depends upon the quantity and integrity of the data used during the learning process
- validating the process is comparable to validating a human being's learning processes.

It is essential for data to be consistent: the principle of the neural network is "if X and Y happen consistently, the output should consistently be Z". If however, X and Y sometimes imply an output of B, or X and F sometimes imply an output of Z, the neural network may not cope with it — just as a child in its learning process. There must also be an adequate

spread within defined operating boundaries of values for X, Y and Z for the neural network to learn properly. Neural networks are very susceptible to the quality of data used in the learning process, and the latter must be expected to be a protracted business. Learning times of weeks or months are not unusual, with vast amounts of data being input to the neural system.

Validation presents a serious problem. It may be possible to validate all or part of the neural software and the hardware on which it is implemented as individual items; the correctness of the learning process is dependent on a variety of variables, at least some of which are essentially subjective. Subjective processes cannot be formally validated. It may be possible to use reverse engineering of a specification to aid this process.

A common use of neural networks is for classification of input patterns into one of a number of output classes. Under certain fairly general conditions, a neural network can perform as a probabilistic classifier, i.e. outputting a probability of an input pattern belonging to one of the output classes. One way of viewing the operation of the neural network is as interpolation in the space defined by its parameters.

The training patterns are the representative examples of classes. After training, previously unseen patterns are classified according to an interpolation between the training examples. The number of neurons in the network determines the complexity of the space in which interpolation is carried out. In this way, with enough neurons, a division of feature-space by arbitrarily complex boundaries can be made, assimilating the fine distinguishing features of the input patterns. Alternatively, by providing only a few neurons, the network is forced to generalise and the boundary values in its training set will be effectively ignored.

Once training is completed the network is switched to a non-learning mode. If the environment ever changes it is up to the designers to switch the network back into learning mode and to re-train it. It has been suggested that, since mechanical systems are subject to degradation, it might be beneficial to leave a neural network in its learning mode so that as the behaviour of the system is it monitoring and/or controlling changes, it can modify its responses. However, leaving a neural network in its learning mode would permit un-predictable behaviour.

Empirical studies [22] have shown that diverse programming invariably does not provide the amount of extra confidence that would be theoretically expected. This is because the types of diversity employed are usually associated with a similar class of standard errors. Conversely, neural networks are sufficiently unique in construction that the situations where an error might be introduced is unlikely to correspond to an equivalent error in a more traditional programming language, and vice versa (see [47]). The same empirical studies have vindicated this in practice.

L.3 Artificial intelligence in safety-related applications

Responsibility: *Software Engineer*

Expert systems are built upon a rigorously defined syntax and their behaviour can therefore be said to be predictable. However, due to their complexity and the way that their knowledge base is often produced by an iterative process, they can produce quite novel views of their environment. It is for this reason that their behaviour can become unreliable.
The unpredictability of the behaviour of a neural network is more fundamental and stems from the way that the network "learns" by modifying its own weights and biases.
Thus any system which is intended to use artificial intelligence should currently incorporate some sort of safety-bag to identify and limit the effects of any unforseen faults.

Appendix M — Dynamic testing

M.1 Dynamic testing

Responsibility: *Software Engineer*

Dynamic test is the conventional method of checking programs. The developer's test strategy for a system should be defined during the specification and design phases, and must include module (i.e. procedure, program, subsystem) functional tests (for normal and error conditions) and module integration tests.

A program under test is executed with differing combinations of input data and the results are analysed to see if they are as expected. This requires the creation of tests, an environment (test harness) to apply the tests to the software being tested, the definition of expected results, a comparison of the actual results to the expected results, a procedure for correcting discrepancies, and a demonstration of test data adequacy (see Section 6.2.3).

A number of commercially available dynamic testing tools are available, but the actual choice may be driven as much by the languages supported and the host machine, as by differences in the functionality of the tool. An emulator may be used to provide a high-level symbolic debug facility, which provides visibility of the software under test. However, with many of these approaches one must be wary of any intrusive effect upon the analysis, since such test harnesses do not test the final executable form, because they have to modify it in order to function. This may have a significant impact on their suitability for testing safety-related systems.

M.2 Black box testing

Responsibility: *Software Engineer*

During black box testing the chosen implementation details are ignored and it is the high-level design that is compared against the requirements. Black box tests, such as a system test (wherein the system under development is simulated) are useful as a preliminary to white box testing. Identifying an error during black box testing will save more in time and costs than if it is not identified until a full white box test.

M.2.1 Low-level testing

This technique of testing should be directed to demonstrating that each component complies with its low-level requirements.

The manageability of testing and the organisation of the software are related. If the software is organised so that test procedures must be run on regions of code with high complexity, the number of required test cases, the complexity of the test procedures, and associated coverage analysis may be unmanageable. The software may have to be organised into small,

functionally isolated components to achieve the necessary test coverage.

Typical errors which may be revealed by this test technique include:

- failure of an algorithm to satisfy the requirements
- incorrect loop operations
- incorrect logic decisions
- failure to process correctly legitimate combinations of input conditions
- failure to process correctly missing or failed input data
- incorrect handling of exceptions, such as arithmetic failures faults or violations of array limits
- incorrect computation sequence
- inadequate algorithm precision, accuracy or performance.

M.2.2 Requirements based tests

If an executable version of the formal specification is available then it is possible to compare the results of testing the full implementation against the results of providing the executable specification with the same inputs. Since executable specifications are not usually very efficient, it is unlikely that the expected test results will be able to be produced at the same time as the testing of the full implementation. This form of testing is a required procedure in DEFSTAN 00-55.

M.3 White or glass box testing

Responsibility: *Software Engineer*

White box testing is also testing against the requirements, but it is performed using a knowledge of the structure of program. Whilst formal proof (see Appendix I.5) and inspection (Appendix N) also use this knowledge they are not usually considered under this heading.

White box testing is concerned with ensuring that all sections of code within the program have been tested, if not all possible paths through it (see Section 6.2.3). Test data is chosen specifically to exercise particular sections of code, and also to check the class boundaries (see Section 6.2.1.3). A quality assurance mechanism can be set up to control the testing, and to quantify what level of testing is required, and what has been carried out. Commercial tools are available to assist in this task.

M.4 Dynamic analysis

Responsibility: *Software Engineer*

The dynamic characteristics of the unit must be tested according to the schematic and timing diagrams. This form of testing is useful to detect errors in time critical applications of the unit. It is also possible to analyse the time spent in each section of the code, thus verifying

the resource loading of the processor.

These tests are carried out by stimulation of input signals from components and by the measuring of logical levels, time behaviour and signal sequences on nodes. They can only be conducted satisfactorily if an operational model and a set of documentation, as well as appropriate tools such as word generator, logic analyser station, etc. are available.

Appendix N — Walkthroughs

A structured walkthrough should form part of an examination and evaluation of the specified functions of the system to ensure that it conforms to, and complies with, the requirements of the specification. It will enable the detection of weak points of understanding concerning the implementation with regard to realisation and use of the product. In contrast to code review, the author is active and the inspector is passive during the walkthrough.

The material should be of a manageable size so that it can be reviewed within a reasonable period of time. Both maintenance standards and user views can be represented together with other interested parties.

Walkthroughs are acknowledged to be an effective process for identifying errors in programs — indeed they can be more effective than computer-based testing for certain types of error. They are particularly important in checking modifications; the making of modifications has been shown by experience to be a far more error-prone process than re-writing the code [6].

N.1 Structured walkthroughs and code inspections

Responsibility: *Software Manager and Software Engineer*

A structured walkthrough, or code inspection, is a general purpose review process in which a designer or programmer leads one or more other members of the design team or other interested parties through a segment of design or code that he or she has written, while the other members or interested parties ask questions and make comments about technique, style, possible errors, violation of development standards and other problems. The optimum number of participants in this process is three to five, of which only one should have been involved in writing the code being reviewed [6]. It should form part of an examination and evaluation of the specified functions of the system to ensure that it conforms to and complies with the requirements of the specification. It will enable the detection of weak points, or doubt concerning the implementation concerning realisation and use of the product. In contrast to code review (see Appendix O), the author is active and the inspector is passive during the walkthrough. It must be stressed that the process must be based on group responsibility, not individual accountability; the latter will merely make the author feel "on trial" and defensive, and has been proved to defeat the process. The review is at its most effective when the author of the program is encouraged by the questions to spot his own mistakes in the code. The process should be one of improvement rather than inquisition.

It is often very revealing to try to read another person’s software. One should be able to relate the code to the design and hence to the specification with little or no difficulty. More often than not this is impossible due to poor documentation, an inadequately defined programming language, poor programming style or all three!

The material should be of a manageable size so that it can be reviewed within a reasonable time. Both maintenance standards and user views can be represented together with other

interested parties.

The process concentrates on error *detection* rather than error *correction*, and it is left to the producer to make sure that each reported item is corrected. The following identifies typical members of the walkthrough process (see [48]):

1) *The moderator* — An independent programmer who oversees the walkthrough process.
2) *The designer* — The programmer responsible for producing the program design.
3) *The coder* — The programmer responsible for translating the design into code.
4) *The tester* — The programmer responsible for writing and/or executing test cases or otherwise testing the product of the designer and coder.

On occasions it may be necessary to divide the role of the moderator into various tasks and into representatives, some of whom may not be programmers, of the different interested organisations. For example (see [49]):

1) *The presenter* — The "owner" of the item and probably the person who produced it.
2) *The co-ordinator* — Someone to organise the walkthrough and chair it. Their task must be to ensure that the process is kept along positive, productive lines (see "egoless" programming in [9]).
3) *The secretary* — Who will ensure that the material is issued beforehand and that the records are taken and submitted to the presenter.
4) *The maintenance representative* — Who represents the people who will one day be responsible for maintaining the item.
5) *The standards bearer* — Who scrutinises the item for adherence to the standards that are applicable.
6) *The user representative* — Who checks that the item conforms to the views of the user (who might be the final user of the system or, in the case of a specification say, the "user" of that specification, namely the designer).
7) Any outsiders who can contribute to the scrutiny.

Usage of walkthroughs should reveal discrepancies between specification and solution. The result is a set of comments upon which the producer is expected to act. The walkthrough also acts as an educational exercise for the project team.

N.2 Fagan

Responsibility: *Software Manager and Software Engineer*

A Fagan inspection is a highly formalised review technique conducted strictly to formal guidelines providing a "formal" audit on quality assurance documents. It is aimed at finding errors and omissions in all phases of the program development and is more effective at detecting errors than less formal review techniques such as peer reviews or walkthroughs. However, there is some ambiguity of terminology — the term "structured walkthrough" can also refer to a very formal design review.

Specifically a Fagan inspection is a formal review:

- of one document against others, for example, with reference to traceability matrices
- that seeks and records errors
- initiates rework as necessary
- is led by an independent moderator
- makes use of error checklists which are based on historical data
- leads to approval of the document based on defined completion criteria
- adds to the inspection database.

The inspection process consists of six phases; Planning, Overview, Preparation, Inspection, Re-work and Follow up:

- planning: appointing the inspection team and sorting out administrative matters
- overview: a brief meeting to introduce inspectors to the item concerned
- preparation: familiarisation, of the document, by the inspectors
- inspection meeting: the actual review process
- re-work: the task of correcting the errors found
- follow-up: checking, by the moderator, of the work and the production of statistics.

Each of these phases has its own separate objective. The complete system development (specification, design, programming and testing) must be inspected.

N.3 Peer reviews

Responsibility: *Software Manager and Software Engineer*

Peer reviews (see Section 2.5) are a much more general human-review approach to the achievement of software quality of a software design team. The overriding principle is one of anonymity. A team, preferably 10 to 20 programmers exchange the programs that they have written, and each member of the team evaluates the program he or she is given. This process is usually aligned with a number of general qualitative questions, which the reviewer answers with a rating (1 – 5, poor – excellent, etc.) After the review, the originators recover the (anonymous) results of the critique, with information that shows how well their program fared against the average, etc. whilst not "pointing any fingers" at individuals. Often individuals are required to submit two examples of their programming, one that they regard as their "best effort", and one other [6]. The aim of the process is to improve the general standard of software quality in an organisation.

Appendix O — Code review

Responsibility: *Software Manager and Software Engineer*

Translation from a *Software Requirements Specification* into the corresponding code has proven to be a non-trivial task, mainly due to the complexity of the system being coded. When this process is compounded with iterative improvements and bug fixes the resultant code may be at great variance with its *Software Functional Requirements*.

It is often assumed that fixing a programming bug will create more accurate software. However, this is unfortunately often not the case and a "fix" may in fact create a another fault, or indeed may not even correct the original fault effectively.

These code reviews and analyses (see Section 2.5) should confirm that the products of the programming process are accurate, complete and can be verified. Primary concerns include the correctness of the code with respect to the requirements and the software architecture. These reviews and analyses are usually confined to the source code.

A review of the code should take place before integration. The objective is to ensure that software design procedures were carried out in accordance with the *Quality Assurance Plan*; to ensure that programming was carried out in accordance with the *Software Requirements Specification*, and the *Software Module Specification*; to identify any inconsistencies with the specifications, and formulate corrective or other actions; to detect and report errors that may have been introduced during the software programming process. A number of programming reviews may be carried out, coinciding with completion of various stages in the programming, e.g. after each module, or a defined group of modules; after completion of any routines identified as critical to the whole; after the assembly of modules into the whole program. The programming review should seek to identify any special areas of difficulty, which may require a particular approach, or requiring particular attention at the testing or verification and validation stage, or which may be viewed as problematic or critical in their relationship with aspects of hardware design. The latter may, for instance, mandate a more rigorous interface with the hardware design team.

Execution of a code review is intended to minimise such problems by using both an independent code reader and also various automatic tools.

In a code review [48], the code reader should not have to refer to the programmer for clarification. If clarification is required, then this should be made through additional program comments (or perhaps re-programming in severe circumstances) and **not** verbally. This will lead to code with a greater maintainability that will have a life-span that outlives the availability of the original programmer. The code reviewer can also check that the mandated programming standards are adhered to.

It should also be clear to the code reader that any relevant programming standards have been adhered to. This may just aid conformance to the organisations' quality plans but may be required for legal and/or commercial reasons.

A programming reviewer or review team may recommend such procedures as a code walkthrough or a Fagan inspection for critical functions in code (see Appendix N).

The topics should include:

a) Compliance with the low-level requirements: The objective is to ensure that the code is accurate and complete with respect to the software low-level requirements.
b) Compliance with the software architecture: The objective is to ensure that the code matches the data flows and control flows defined in the software architecture.
c) Ability to be verified and/or validated: The objective is to ensure that the code does not contain statements or structures that cannot be verified and/or validated (for example, code that must be altered in order to test it).
d) Conformance with standards: The objective is to ensure that the programming standards have been followed during the development of the code and that deviations have been justified. Special attention should be given to restrictions on software complexity, including the degree of coupling between software components, the nesting levels for control structures and the complexity of logical or numerical expressions.
e) Traceability: The objective of this analysis is to ensure that the software architecture and the low-level requirements are traceable to the source code.
f) Accuracy and consistency: The objective of these analyses should be to consider stack usage, fixed point arithmetic overflow and resolution, resource contention, worst case execution timing, exception handling, identification of variables that are used but not set, identification of variables that are not used, data corruption due to task or interrupt conflicts, etc.

Static verification of code does not require the program to be executed. It can consist of one or more of the following; a formal inspection process, use of a static program analyser and formal proof arguments. Static analysis is highly recommended and software engineering tools are available that not only perform checks on the structure of the code, but can also assist in proving its conformance to the specification (see Appendix J). Use of static analysis tools will enable consideration of issues such as stack usage, fixed point arithmetic overflow and resolution, resource contention, worst case execution timing, exception handling, identification of variables that are used but not set, identification of variables that are not used, data corruption due to task or interrupt conflicts, array bound violations, etc. This is a list of those areas that are normally associated with errors even when a conscientious programmer has been employed.

At the end of this review, the reviewer's comments should be recorded in the *Code Review Results*, which is passed to the software quality team, and the resulting actions followed through and documented under full change control. The value of this approach should not be underestimated. However, because important omissions or errors may be overlooked in a complex system, it must not be totally relied upon.